Aprenda a programar com Python

Descomplicando o desenvolvimento de software

Casa do Código | alura

LEONARDO SOARES E SILVA
GABRIEL FORTES

Sumário

- ISBN
- Prefácio
- Sobre este livro
- Primeiros passos na programação
  - 1 A importância dos softwares em nosso dia a dia
  - 2 Resolvendo problemas com a programação
  - 3 Armazenando dados em nossos programas
  - 4 Criação de variáveis
  - 5 Erros… eles acontecem e são importantes
  - 6 Leitura e apresentação de dados
- Estruturas condicionais e de repetição
  - 7 Controlando o fluxo de execução do código com condições
  - 8 Repetição de instruções
- Organizando o código com funções
  - 9 Funções no Python
- Estruturas para organização de dados
  - 10 Listas e Tuplas
  - 11 Conjuntos, Dicionários e Matrizes
  - 12 Tópicos intermediários em programação
  - 13 Referências

ISBN

Impresso: 978-85-5519-301-9

Digital: 978-85-5519-300-2

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Silva, Leonardo Soares e
Aprenda a programar com Python : descomplicando o desenvolvimento de software / Leonardo Soares e Silva, Gabriel Fortes. -- São Paulo : AOVS Sistemas de Informatica, 2022.

ISBN 978-85-5519-301-9

1. Python (Linguagem de programação para computadores) 2. Software - Desenvolvimento I. Fortes, Gabriel. II. Título.

22-107801 CDD-005.133

Índices para catálogo sistemático:

1. Python : Linguagem de programação : Computadores : Processamento de dados 005.133

Maria Alice Ferreira - Bibliotecária - CRB-8/7964

Prefácio

Se há poucos anos as competências de programação eram necessárias apenas para quem se queria dedicar à informática, hoje são requisito importante em muitas outras atividades. Para além da capacidade de criar programas utilizando uma dada linguagem de programação, aprender a programar contribui para o desenvolvimento de competências importantes noutros domínios, como as habilidades relacionadas às estratégias de resolução de problemas.

A linguagem Python ganhou muita relevância nos últimos anos na indústria de software, sendo hoje utilizada em projetos de muitos tipos, desde os mais simples até os mais sofisticados, por exemplo, na área da Inteligência Artificial. Ainda assim, comparado a outras linguagens de programação, Python apresenta uma sintaxe mais simples, ou menos complexa, tendo, por isso, sido adotada nas unidades curriculares introdutórias de programação em muitas instituições de ensino superior de todo o mundo.

A aprendizagem introdutória de programação exige uma abordagem muito prática, envolvendo o desenvolvimento de um elevado número de programas de complexidade crescente. É uma tarefa complicada para muitas pessoas, dado que, para além do conhecimento da sintaxe e da semântica da linguagem de programação, é necessário desenvolver estratégias de resolução de problemas adequadas aos desafios que são colocados.

É nesse contexto que materiais pedagógicos de qualidade, que apresentem os conceitos e exemplos de uma forma simples e direta, podem ser um importante suporte à aprendizagem. É esse o caso do presente livro, "Aprenda a programar com Python: Descomplicando o desenvolvimento de software.", de autoria do Dr. Leonardo Soares e Silva e do Dr. Gabriel Fortes, editado pela Casa do Código. Este livro resulta da experiência dos autores em disciplinas introdutórias de programação, bem como do seu envolvimento em atividades de

investigação que buscam formas de suportar eficazmente a aprendizagem dos estudantes. O livro será certamente útil para pessoas com pouco ou nenhum conhecimento prévio em programação, ou seja, aquelas que estão vivenciando a sua primeira experiência neste domínio.

O livro usa uma linguagem simples e direta, procura estimular estratégias de aprendizagem adequadas à programação, sendo muito apoiada em exemplos do dia a dia da leitora, do leitor, ilustrando como conceitos da programação são utilizados em diversos tipos de programas, como nas redes sociais ou nos jogos. A abordagem utilizada recorre a recursos visuais para fazer analogias ou explicar conceitos da programação que são naturalmente abstratos. O livro conta igualmente com uma grande variedade de exercícios com respostas que vão ajudar você a desenvolver as suas próprias competências de programação. O meu conhecimento dos autores e do próprio livro levam-me a recomendar a sua utilização por todas as pessoas que se pretendam iniciar no mundo fascinante, mas complexo, da programação de computadores.

Dr. António José Mendes, Professor no Departamento de Engenharia Informática da Universidade de Coimbra - Portugal.

Sobre este livro

Seja bem-vindo, leitor, seja bem-vinda, leitora. Ao longo deste livro vamos apresentar o incrível universo da programação e esperamos que você aproveite a leitura para explorar o seu lado criativo na criação de programas!

A proposta deste livro é promover o aprendizado de programação utilizando uma abordagem pedagógica, centrada na pessoa que nunca programou (ou que teve pouco contato com a área). Ao longo de nossa experiência enquanto professores, pudemos observar e entender as principais dificuldades dos aprendizes de programação e aqui você poderá aproveitar toda essa dinâmica de estudo que pudemos aprimorar.

Os conceitos de programação apresentados neste livro são fundados no Python 3.10, mas em sua maioria aplicam-se às versões anteriores da versão 3.

Sobre os autores

Leonardo Soares e Silva é professor de computação do Instituto Federal de Pernambuco há mais de 10 anos. Também é pesquisador e investiga estratégias para facilitar os processos de ensino e aprendizagem de programação. É doutorando pela Universidade de Coimbra (Portugal) e possui mestrado pela Universidade Federal de Campina Grande.

Gabriel Fortes é professor e pesquisador na Universidad Alberto Hurtado (Chile), onde faz investigações sobre pensamento, discurso e espaços educacionais pensados para promoção de aprendizagem colaborativa. É mestre e doutor pela Universidade Federal de Pernambuco em Psicologia Cognitiva.

Como ler este livro

Olá, em breve você será apresentada(o) ao incrível universo da programação, mas antes de continuar precisamos dar algumas dicas sobre como ler este livro. É muito importante você dedicar alguns minutos para ler essa seção, pois ela vai ajudar você a compreender melhor a lógica por trás da construção de cada capítulo.

Se você é novato ou novata na programação ou não conhece a linguagem Python, recomendamos ler este livro de forma progressiva e sequencial. Ou seja, iniciar no capítulo 1, seguir para o 2, e assim em diante. Caso já saiba algo, uma recomendação sempre válida é revisar os conteúdos "já conhecidos" antes de avançar para seções que vão lhe apresentar novo conhecimento.

Independente de qual seja a forma como vai ler este livro, é imprescindível que você realize as atividades propostas, para praticar o que foi apresentado. Sabe-se que a prática é fundamental ao aprendizado de programação, uma área que envolve conceitos teóricos, mas também muitas ações práticas aplicadas na construção de softwares. A realização das atividades também é útil para que você avalie o que sabe e o que ainda não sabe, podendo direcionar melhor os seus estudos.

A seguir destacamos alguns elementos visuais que utilizaremos ao longo do livro para auxiliar sua aprendizagem.

Ícone utilizado no livro

Para destacar informações importantes no capítulo utilizamos o ícone a seguir. Fique atento(a), pois são revisões de tópicos relevantes e avisos sobre erros comuns na programação e como evitá-los.

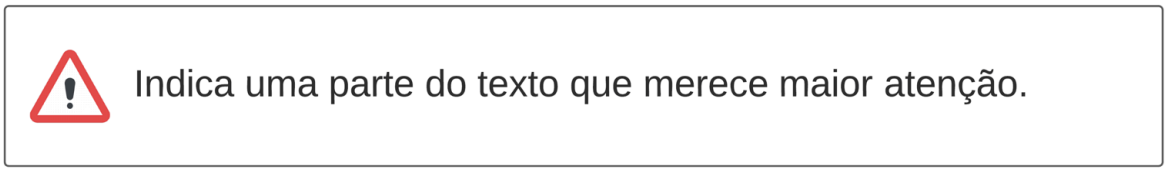

Figura 0.1: Este ícone indica uma parte do texto que merece maior atenção.

Códigos de programação

Os códigos de programação serão apresentados em uma fonte diferente dos demais textos do livro, como representado a seguir:

```
# Exemplo de algoritmo.
```

Recomendamos que antes de continuar a leitura dos demais capítulos instale as ferramentas necessárias para utilizar o Python. Esse procedimento é necessário para que você possa testar os códigos de programação apresentados neste livro e assim possa visualizar o resultado produzido. Caso não saiba como instalar o Python, no Capítulo 1 apresentamos o procedimento.

Seção inicial

Em cada capítulo do livro nós teremos uma seção onde apresentaremos três pontos importantes para o seu aprendizado: 1. O que você já sabe? 2. O que veremos? 3. Por que é importante?

A primeira pergunta é para você lembrar alguns conceitos que já sabe e que são necessários para o capítulo. A segunda pergunta apresenta um resumo do que será visto no capítulo e, por fim, a terceira pergunta destaca por que é importante aprender os conceitos que serão apresentados.

O que você vai aprender

Nesta seção do capítulo vamos destacar o que esperamos que você aprenda após terminar sua leitura do capítulo. É importante criar

essa consciência para que você saiba quais são os objetivos que devem ser alcançados. Isso vai ajudar você a direcionar seu estudo sobre os pontos mais importantes. Também é importante que ao término do capítulo você releia os objetivos e reflita se os alcançou. Aprender através de um livro implica no desafio de ser disciplinado, assim, estabelecer e avaliar objetivos é um ótimo jeito de dar o primeiro passo para que a aprendizagem seja efetiva.

Sumário

Ao término de cada capítulo incluímos um sumário que vai ajudar a relembrar as partes importantes do capítulo e avaliar se algum conhecimento requer ser mais aprofundado.

Mapa mental

Por fim, utilizamos um mapa mental que é uma representação gráfica com os principais tópicos do assunto. Vale a pena usar essa figura para compreender como os conceitos se relacionam, como também avaliar seu entendimento sobre eles. Boa leitura! :)

Primeiros passos na programação

Capítulo 1
A importância dos softwares em nosso dia a dia

1. O que você já sabe?

Os softwares estão cada vez mais presentes em nosso dia a dia e apresentam forte tendência de crescimento para os próximos anos. A criação de aplicativos representa uma ótima oportunidade profissional e também de liberdade, pois permite a criação de soluções tecnológicas para os problemas do nosso dia a dia.

2. O que veremos?

Neste capítulo, começaremos a nossa jornada apresentando conceitos básicos da área de programação e que serão fundamentais no decorrer do livro. Aqui é o nosso ponto de partida!

3. Por que é importante?

Aprender a programar é um universo novo para a maioria das pessoas. Há vários conceitos que precisam ser apresentados antes de começarmos a criar nossas próprias aplicações. Logo você aprenderá que a programação é uma pirâmide de conceitos, sendo importante você construir uma base sólida de conhecimentos.

O que você vai aprender

Ao término deste capítulo você compreenderá:

1. Por que a computação e a área de programação são importantes para a sociedade;

2. Como os programas são criados;
3. Conceitos-chaves da área:
    i. Algoritmos;
    ii. Linguagens de programação;
    iii. Sintaxe;
    iv. A linguagem Python.
4. Como realizar a instalação do Python.

1.1 A transformação digital em nossas vidas

Nossa vida é rodeada de aplicativos. Pela manhã nós somos acordados por um despertador em nosso *smartphone*; utilizamos o WhatsApp para falar com amigos e utilizamos *e-mail* para troca de arquivos do trabalho. Não há dúvidas de que os softwares são parte fundamental do nosso dia a dia e atualmente é difícil pensar em uma realidade sem essa tecnologia.

Podemos ir um pouco além e pensar como a tecnologia também vai transformar o nosso futuro. Você já deve ter ouvido falar dos carros autônomos que não precisam de motorista. Eles existem, pois computadores e sensores eletrônicos são embutidos nos veículos. Esses equipamentos analisam a distância para o veículo da frente, a presença de pedestres ou mesmo em qual cor que está o semáforo. Todo esse processo que parece muito distante já é realidade e é possível pela existência de *softwares* que foram programados para realizar essas ações.

Os softwares existem com o objetivo de resolverem nossos problemas. A boa notícia é que as expectativas são de expansão deste setor, pois há uma forte busca das empresas por melhorias em seus processos produtivos e redução de custos. Além disso, cada vez mais pessoas estão com acesso a computadores e smartphones. Fatores como esses fazem com que os profissionais da área de desenvolvimento de software sejam um dos mais requisitados no

mercado de trabalho (BRASSCOM, 2019). Afinal, ao dizer que precisamos de aplicativos para solucionar problemas, também estamos falando em demanda por pessoas qualificadas para desenvolvê-los.

Antes de pensar em criar os seus próprios aplicativos é preciso compreender alguns conceitos. Um software também pode ser conhecido por aplicação ou programa, utilizaremos todas as palavras para nos referir ao mesmo recurso. Mas afinal, **o que seria um aplicativo?** Segundo o dicionário Oxford Languages, sua definição é:

> "um programa é concebido para **PROCESSAR DADOS** eletronicamente, **facilitando** e **reduzindo** o tempo de execução de uma tarefa pelo **usuário**."

Destacamos o papel dos aplicativos em processar dados (informações) para promover ganhos para um usuário (pessoas ou empresas). No exemplo que apresentamos sobre o carro autônomo, as informações coletadas no entorno do veículo (distância para outros veículos, pedestres próximos, entre outros) são dados que serão processados com o objetivo construir veículos que vão facilitar a nossa vida e reduzir acidentes. Assim, podemos dizer que os aplicativos existem com um único propósito: **resolver nossos problemas utilizando o incrível poder dos computadores**.

Podemos dizer que o grande proveito da computação está em aproveitar os seus recursos que permitem o processamento de milhões de dados em pouco tempo, na sua capacidade de comunicação que possibilita a interação com outras pessoas (ou computadores) em qualquer parte do planeta, e na possibilidade de executar tarefas por longos períodos de tempo (24 horas por dia), que em nós causaria, facilmente, uma lesão por esforço repetitivo (LER).

Neste capítulo faremos uma breve apresentação de elementos importantes da programação. A leitura será útil para você conhecê-los e se familiarizar com alguns termos que utilizaremos no decorrer do livro.

1.2 Como os programas são criados

Programar é controlar o computador para que ele realize ações para nós. Podemos fazer com que ele lembre de informações para nós, como um lembrete em uma agenda, e dizer para que em um horário em específico nos avise desse compromisso. Para criar aplicações como essa é preciso de uma forma de comunicação com o computador e isso é feito por meio de uma linguagem especial conhecida por **linguagem de programação**.

Ao longo do livro vamos aprender esta nova língua, o que vai demandar da sua parte memorização para lembrar das novas "palavras" que você vai aprender. Com elas você poderá instruir o computador a fazer as mais variadas ações. Essa comunicação é realizada de uma forma escrita, ou seja, o programador escreve um conjunto de **instruções** utilizando a linguagem de programação, que serão lidas pela máquina que realizará às ações demandadas.

Os computadores podem ser programados para realizar uma série de ações ou responder a determinados eventos quando eles acontecerem. Vamos tomar como exemplo jogos de tiro, como Fortnite, CSGO, Call of Duty, entre outros. Os programadores desses jogos escreveram instruções para que o computador responda às ações do jogador, como um clique no mouse ou toque em uma tecla no teclado. Seguindo a sua programação, o computador oferece uma resposta, que pode ser a apresentação de uma animação no monitor, reprodução de um som, entre outros.

1.3 Conhecendo os algoritmos

Um conceito importante para a programação são os **algoritmos**. Podemos pensar neles como uma forma de estruturar as ações necessárias para alcançar um determinado objetivo. Por exemplo, a funcionalidade de atirar em um jogo, como o Fortnite, é na realidade formada por um conjunto de ações:

1. identificar quando o clique no mouse ocorrer;
2. diminuir uma bala;
3. verificar se atingiu inimigo;
4. diminuir sangue do inimigo.

Um dos papéis da pessoa programadora é identificar quais as ações precisam ser realizadas para que uma funcionalidade seja criada. A esse processo dá-se o nome de **pensamento algorítmico**, que envolve os seguintes passos:

- Compreender o problema que se espera resolver;
- Quebrar o problema em partes menores e mais simples;
- Escrever as instruções de programação para resolver cada uma dessas partes;

O exemplo do Fortnite ilustra como as ações acima ocorrem. Compreendemos o problema que seria resolvido (como realizar um tiro com uma arma no jogo) e quebramos em partes menores (reconhecer o momento em que o jogador deseja atirar, diminuir a bala, verificar se atingiu inimigo, entre outros) que podem ser resolvidos com instruções de programação.

Podemos fazer uma analogia entre um algoritmo e uma receita de bolo. Em ambos, há uma série de instruções que precisam ser realizadas em uma sequência correta para que o resultado seja alcançado satisfatoriamente. Qualquer falha em uma das ações ou mesmo executá-las fora de ordem poderá comprometer o resultado final. Veja a seguir um exemplo de algoritmo para trocar uma lâmpada queimada:

1. Comprar a lâmpada;
2. Montar a escada;
3. Subir na escada;
4. Retirar a lâmpada queimada;
5. Colocar a lâmpada nova.

Observe que há um conjunto de ações (no exemplo acima são 5) que são realizadas sequencialmente, uma após a outra para alcançar um objetivo (acender uma lâmpada).

Como falamos anteriormente, é importante compreender que a ordem das ações influenciará o resultado final. Observe que no algoritmo a seguir há um problema:

1. Comprar a lâmpada;
2. Subir na escada;
3. Montar a escada;
4. Retirar a lâmpada queimada;
5. Colocar a lâmpada nova.

Perceba que as ações 2 e 3 estão invertidas, pois não é possível subir na escada antes de tê-la montado corretamente. Fique atento na hora de construir seus algoritmos, pois eles produzirão respostas incorretas se as ações estiverem fora de ordem.

Na prática, um algoritmo computacional é apenas uma representação de como resolvemos um determinado problema. É importante saber seus conceitos, como a necessidade de pensar as ações de forma sequencial, que serão materializados pelo programador por meio das instruções que ele/ela vai escrever.

Já deu para perceber que um programa é formado por palavras, como em um livro. O computador lê sequencialmente cada uma das linhas do **código de programação** e executa suas ações. Essa leitura é parecida como lemos um livro, inicia de cima para baixo e da esquerda para direita.

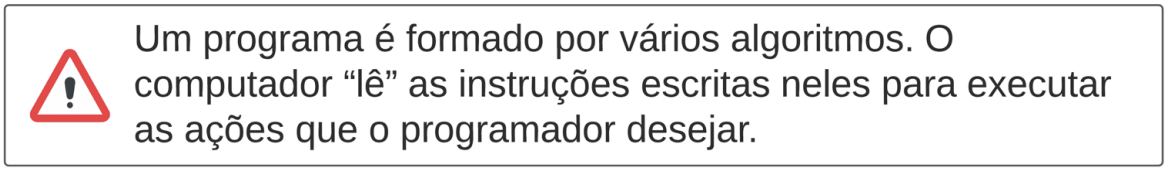

Figura 1.1: Um programa é formado por vários algoritmos. O computador “lê” as instruções escritas neles para executar as ações que o programador desejar.

O que o computador executou nas linhas anteriores influenciará a execução das próximas. No exemplo do Fortnite, a segunda ação realizada é a diminuição da bala na arma do jogador, mas isso somente fez sentido após a identificação de que o mouse foi clicado.

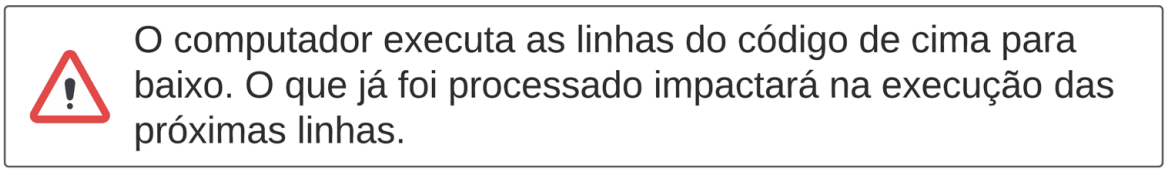

Figura 1.2: O computador executa as linhas do código de cima para baixo. O que já foi processado impactará na execução das próximas linhas.

1.4 Entrada e saída de dados

Em geral, um algoritmo depende de alguma informação para poder iniciar o seu **processamento** (execução das instruções), e chamaremos isso de **entrada**. No algoritmo que apresentamos do Fortnite, o clique do mouse é a nossa entrada, pois é ele quem desencadeará a execução das demais instruções.

Um outro conceito importante é que os algoritmos também produzem uma **saída**. Ela é o resultado obtido ao término da

execução das instruções. No exemplo do Fortnite, a nossa saída é o tiro realizado.

É fundamental compreender o conceito de entrada, processamento e saída. Para isso apresentaremos uma analogia com a criação de um suco, que é o resultado das ações feitas para transformar a fruta em líquido.

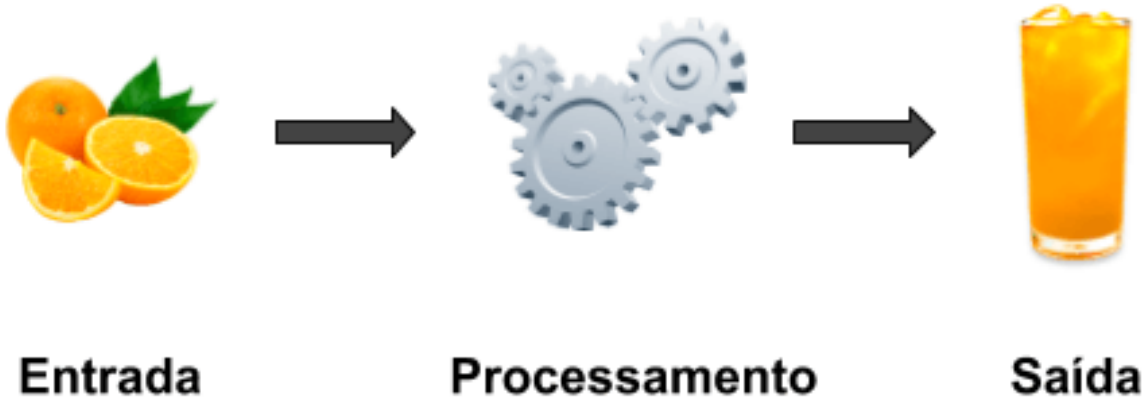

Figura 1.3: Transformação de uma laranja em suco.

A depender da entrada informada ao algoritmo o seu resultado será diferente. Ainda que as ações sejam as mesmas, pela mudança da fruta obtemos um suco diferente.

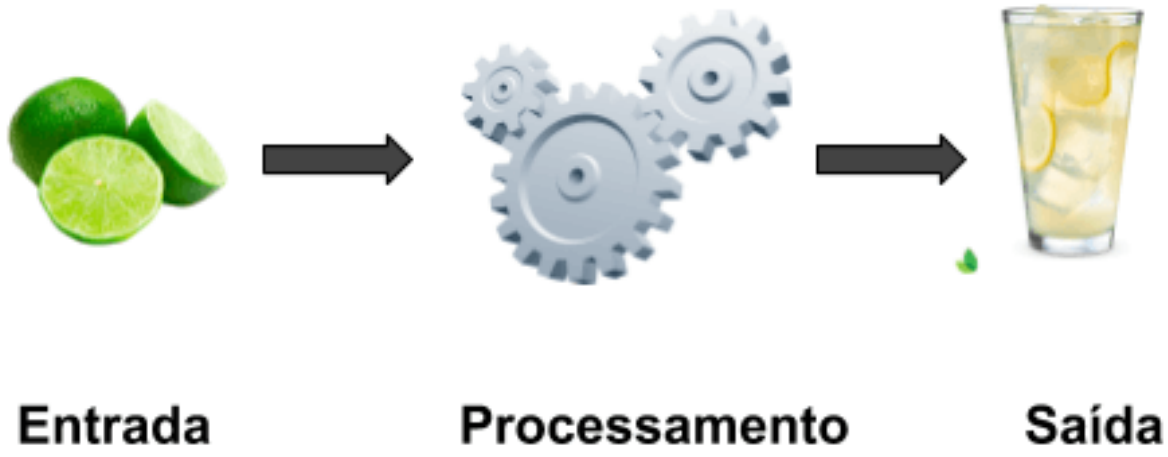

Figura 1.4: Transformação de um limão em suco.

Com frequência você criará algoritmos que vão receber dados de entrada por meio de dispositivos, como teclado ou mouse. A saída é normalmente apresentada em um monitor, ou em uma impressora.

> ⚠ Os programas processam entradas que resultam em saídas.

Figura 1.5: Os programas processam entradas que resultam em saídas.

1.5 Vamos de Python!

Existem várias linguagens de programação disponíveis e não podemos afirmar que há uma melhor. Neste livro apresentamos uma linguagem de programação de nome Python, pois é uma das mais adequadas para o aprendizado de programação. Um dos motivos é a facilidade em instalar os softwares necessários para programar nesta

linguagem (sim, para criar programas é preciso utilizar outros programas) e da forma mais simplificada para a escrita do código.

Ao mesmo tempo, isso não quer dizer que seja uma linguagem limitada ou de iniciantes apenas. O Python é utilizado por grandes empresas de tecnologia como o Google, Instagram e Facebook e atualmente é uma das principais linguagens de programação do mercado com inúmeras ofertas de trabalho (REALPYTHON, 2020). Essa linguagem possibilita a criação de aplicativos para computador, Internet, dispositivos móveis (smartphones), jogos, entre outros. Sem dúvidas, aprender essa linguagem será de grande utilidade no seu futuro profissional.

Ao longo do livro você aprenderá sobre os recursos oferecidos pelo Python e quais instruções devem ser escritas para utilizá-los. Antes, é preciso saber que existem regras a serem seguidas nesta escrita e que recebem o nome de **sintaxe**. Sua memória terá um papel importante para que você possa lembrar da sintaxe correta para uma determinada instrução.

O conceito de sintaxe não é exclusivo do Python e cada linguagem de programação apresenta a sua própria. No futuro, se precisar mudar de linguagem basta aprender a sua sintaxe, pois os conceitos, em sua maioria, serão os mesmos.

O processo de criação de aplicativos

Agora que sabemos que os programas são formados por palavras (instruções), escritas em uma linguagem de programação (Python) e seguindo regras específicas (sintaxe), vamos entender como isso acontece na prática.

O processo de criação de aplicativos envolve, basicamente, a escrita dos nossos algoritmos e uma forma de executá-los. Podemos utilizar qualquer editor de textos para escrever códigos Python, como o bloco de notas do Windows. No entanto, em geral utilizamos editores especializados, como o Microsoft Visual Studio Code

(https://code.visualstudio.com/), que é gratuito, e oferecem recursos para apoiar a criação de programas.

Além de um editor, precisamos de um programa que será utilizado para a execução do nosso código de programação. Pense nele como um tradutor, que entende a linguagem de programação e a traduz para a linguagem de máquina, o formato realmente compreendido pelo computador. No Python esse procedimento é feito por meio de um interpretador que precisa ser instalado. Ele também tem como objetivo verificar se o nosso código segue a sintaxe corretamente, apresentando um aviso em caso de problema.

O interpretador está disponível para múltiplos sistemas operacionais (Windows, macOS, Linux, entre outros) e pode ser descarregado pelo site oficial no endereço https://www.python.org/downloads/. Ao acessar esse link você encontrará duas versões do Python, que no momento da escrita deste livro estão em 3.10.2 e 2.7. Recomendamos que instale a versão 3+, pois a versão 2+ existe para fins de suporte legado e deixou de receber atualizações. Além disso, como este livro foi escrito direcionado ao Python 3+, pode ser alguns dos recursos não funcionem no Python 2+.

Há diversas opções para a instalação no Windows e recomendamos o uso do *Windows Installer*, que pode ser 32 ou 64bits a depender do seu sistema operacional. Após a finalização deste processo o instalador criará uma entrada no Menu Iniciar onde é possível acessar o *Python Idle Shell*. Esse programa possibilita a escrita de códigos de programação diretamente nele, como também a execução de códigos escritos em outros editores. Para este último caso, você deve acessar o menu *File* e em seguida clicar em *Open*. Após isso, escolha o arquivo que contém os códigos de programação (extensão .py) e ele será executado.

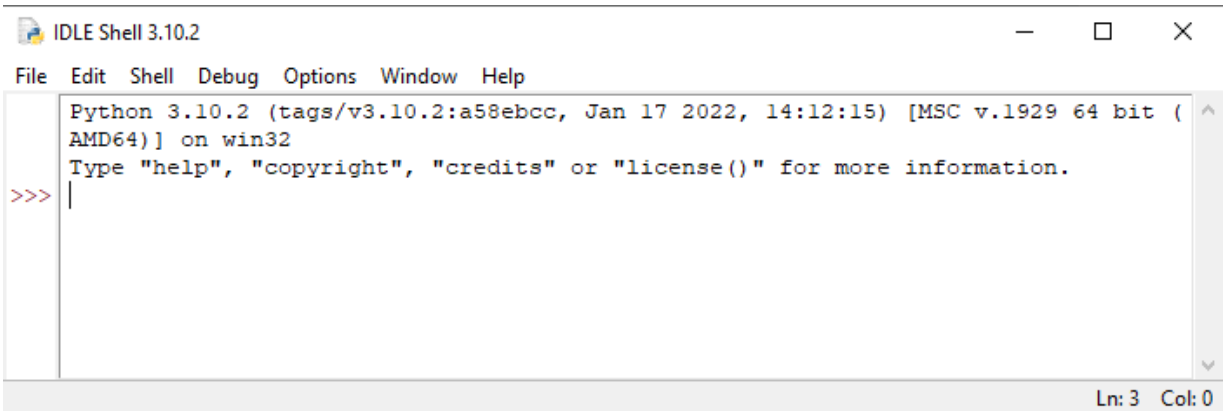

Figura 1.6: Interface do Python Idle Shell.

A instalação no macOS também é feita por um instalador próprio (extensão .pkg) localizado no mesmo site. Para o Linux deve-se consultar os repositórios oficiais da distribuição onde o Python será instalado. Em ambos os sistemas operacionais (macOS e Linux) há uma grande chance de você já possuir uma versão do Python instalada por padrão. Mesmo que ela seja inferior à 3.10, ainda assim é útil para iniciar os estudos.

Há também uma terceira opção, caso não consiga instalar o Python em seu computador. São sites que disponibilizam uma versão online do interpretador, como o *Replit* (https://replit.com/languages/python3). Eles são úteis neste início, mas recomendamos fortemente que instale o Python em sua máquina.

Agora que você sabe o básico sobre os programas, vamos iniciar nossa jornada de programação! :)

Questões

1) O que são aplicativos?

2) Cite dois exemplos de dispositivos de entrada e saída para o computador.

3) O que é a sintaxe de uma linguagem de programação?

4) O que são algoritmos de computação e quais as regras existentes para a sua criação?

5) Para que utilizamos as instruções em programação?

6) Por que a programação é uma importante área para a nossa sociedade atual?

7) Por que precisamos da linguagem de programação?

Respostas

1) O que são aplicativos?

R: Aplicativos são ferramentas projetadas para resolverem problemas do nosso dia a dia. Para isso, eles utilizam os incríveis recursos dos computadores (elevado poder de processamento e armazenamento de dados) para alcançar seus objetivos.

2) Cite dois exemplos de dispositivos de entrada e saída para o computador.

R: Entradas: mouse e teclado. Saída: Monitor e impressora.

3) O que é a sintaxe de uma linguagem de programação?

R: Sintaxe é a forma como escrevemos as instruções de programação de uma forma que o computador consiga entendê-las para realizar as ações aos quais foram programados.

4) O que são algoritmos de computação e quais as regras existentes para a sua criação?

R: Algoritmos são um conjunto de instruções de programação que serão lidas pelo computador que executará o que está escrito. Algumas regras são, as instruções devem ser escritas na sequência correta e devem almejar alcançar algum objetivo.

5) Para que utilizamos as instruções em programação?

R: Utilizamos as instruções de programação para fazer com que o computador realize ações para nós.

6) Por que a programação é uma importante área para a nossa sociedade atual?

R: Há uma tendência de informatização de diversos setores da sociedade. Isso é impulsionado pela necessidade de reduzir custos, aumentar a eficiência e diminuir erros, fatores em que os computadores são bons em alcançar.

7) Por que precisamos da linguagem de programação?

R: O computador não entende o nosso idioma, mas sim uma linguagem especial conhecida por Linguagem de Programação. Com ela, nós escrevemos instruções de programação, seguindo uma sequência lógica conhecida por algoritmos.

1.6 Resumo e mapa mental

- Neste capítulo aprendemos que os softwares estão presentes em nosso dia a dia e a tendência é que o seu uso seja ampliado nos próximos anos;
- O papel da pessoa programadora é escrever algoritmos que vão controlar recursos do computador;
- Utiliza-se uma linguagem de programação para a escrita dos algoritmos;
- Aprendemos como realizar a instalação do interpretador do Python.

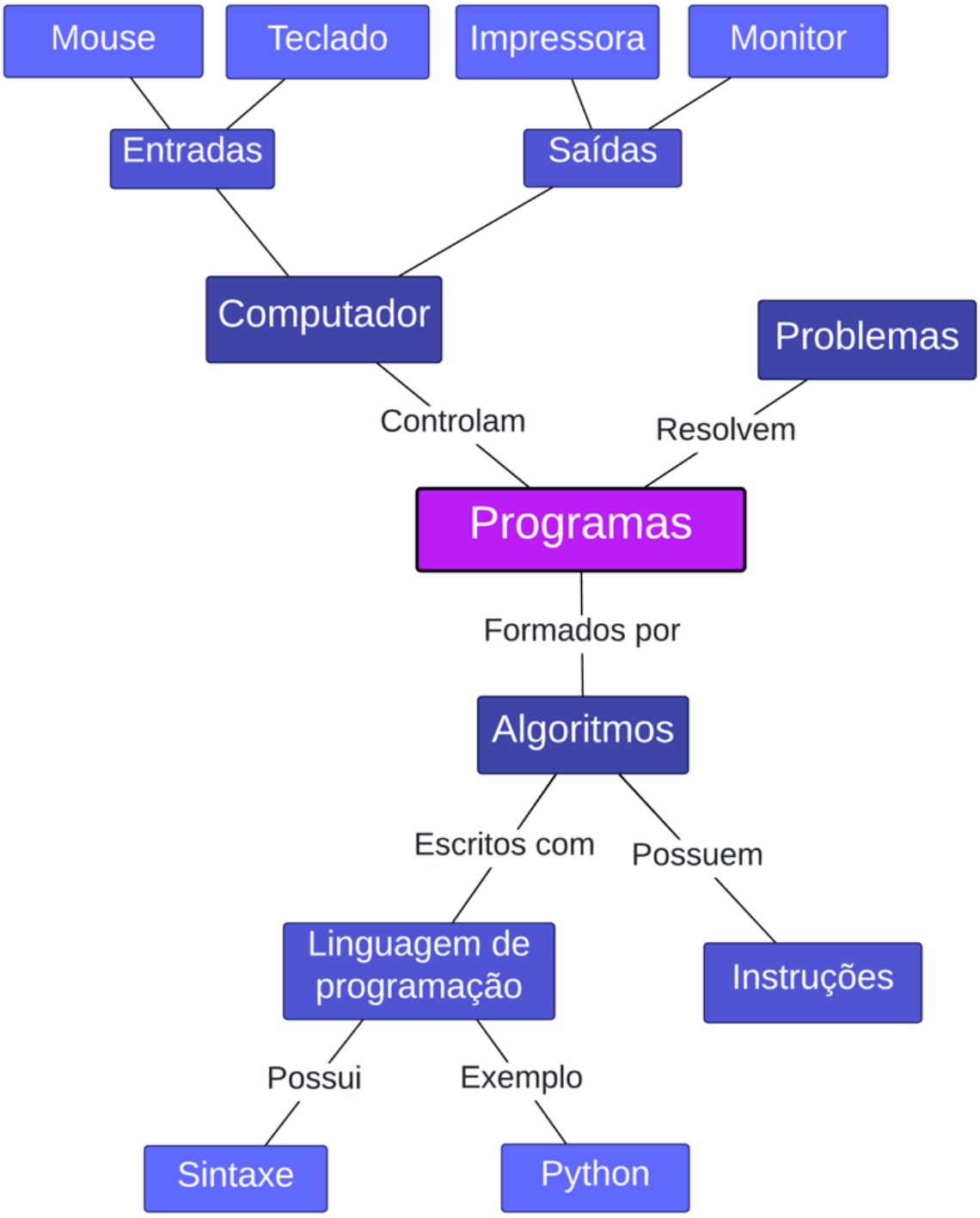
Mouse
Teclado
Impressora
Monitor
Entradas
Saídas
Computador
Problemas
Controlam
Resolvem
Programas
Formados por
Algoritmos
Escritos com
Possuem
Linguagem de programação
Instruções
Possui
Exemplo
Sintaxe
Python

Figura 1.7: Mapa mental do capítulo 1.

Capítulo 2
Resolvendo problemas com a programação

1. O que você já sabe?

Os programas existem para resolver problemas de pessoas e empresas. Seu processo de criação demanda a escrita de instruções utilizando uma linguagem de programação. É por meio dessa linguagem que conseguimos nos comunicar com o computador e definir o que ele deve realizar para nós.

2. O que veremos?

A base da programação é resolver problemas. Pense nas funcionalidades de um software como um conjunto de soluções para diversos problemas. Assim, para construir essas soluções é preciso ter um entendimento detalhado sobre o problema que será atacado.

Neste capítulo vamos apresentar estratégias que podem ser usadas para tornar a resolução de problemas um processo mais eficaz.

3. Por que é importante?

Entender corretamente o problema que será trabalhado é fundamental para construir uma solução adequada. Infelizmente, muitos aprendizes falham nessa etapa e como consequência criam programas incorretos. Felizmente, existem estratégias que ajudam na resolução de problemas e dedicamos um capítulo exclusivamente para elas.

O que você vai aprender

Ao término deste capítulo você compreenderá:

1. Como realizar o processo de identificação de um problema;

2. Quais técnicas podem ser utilizadas para esboçar a solução de um problema;
3. A importância das ações de resolução de problemas para a criação de algoritmos.

2.1 Etapas para a solução de um problema

Programar é resolver problemas do dia a dia com o uso de computadores. Pense no *WhatsApp*, esse aplicativo de sucesso soluciona um problema comum à maioria das pessoas: a necessidade de comunicação.

Sem a capacidade de entender os problemas existentes na comunicação entre as pessoas, antes do *WhatsApp*, seus desenvolvedores não seriam capazes de entregar o conjunto de funcionalidades que fazem deste software um líder de mercado: simplicidade, amplo acesso e praticamente sem custos.

Não adiantar conhecer os recursos da linguagem de programação se você não consegue resolver problemas de maneira consistente e, mais importante, de forma eficaz. Essa é uma parte negligenciada por muitos aprendizes que, na vontade de aprenderem a programar, preferem os conceitos técnicos. Em nossos anos como professores observamos isso com frequência, a consequência é que essas pessoas terão dificuldades para criar aplicações adequadamente, pois não entendem corretamente qual o problema que estão tentando resolver (PRATHER et al., 2019). Em razão disso, acreditamos ser fundamental apresentar a você técnicas de resolução de problemas, antes de explorarmos o universo da programação.

Existem diversas estratégias que podem ser ensinadas e treinadas para auxiliar o processo de resolução de problemas. Fundamentamos nossa proposta de trabalho na metodologia de

Gomes e Mendes, que estabelece práticas sólidas adaptadas à área de programação (GOMES e MENDES, 2007). De acordo com esses autores, o processo de solução de um problema envolve a realização de seis etapas: 1) entender o problema; 2) caracterizar o problema; 3) representar o problema; 4) resolver o problema; 5) refletir na solução e 6) comunicação a solução. Para nós, neste livro, interessam apenas as cinco etapas que serão descritas ao longo deste capítulo.

Entender o problema

Devemos pensar em um programa como um conjunto de funcionalidades que nos possibilita realizar algo. Por exemplo, se no passado a escrita era feita em uma máquina de datilografar, com o avanço da computação e do surgimento de editores de texto, como o *Word*, passamos a explorar a escrita de documentos eletrônicos. Essa tecnologia resolveu problemas que existiam à época, em especial as dificuldades para formatação dos textos.

Percebe-se que o sucesso de qualquer tecnologia requer o entendimento sobre o problema que pretende ser resolvido. No exemplo anterior, a substituição das máquinas de datilografia somente ocorreu, pois entendeu-se quais eram as dificuldades dos seus usuários.

Muitas vezes o problema não está explícito e requer a leitura cuidadosa e interpretação dos materiais que o descrevem, sejam textos, diagramas ou outras informações. Trazendo para o nosso contexto, como aprendiz, você lerá enunciados de exercícios que descrevem uma situação e pedem para que você crie algoritmos para solucioná-las. Veja um exemplo a seguir:

Leia quatro números do teclado que nomearemos como A, B, C e D, respectivamente. A seguir, calcule o produto de A e B, chamaremos de X; e o produto de C e D, que chamaremos de Y. Por fim, subtraia Y de X e apresente o resultado no monitor do usuário.

Um erro no entendimento das operações matemáticas desse enunciado fará com que o seu software calcule a resposta incorreta.

Muitos aprendizes não dedicam tempo suficiente à leitura e interpretação do problema. Muitos dos nossos alunos e alunas relatam que preferem seguir para a criação de códigos e desenvolver seu entendimento do problema ao mesmo tempo em que programam. Porém, sabe-se que realizar essas duas ações simultaneamente não é uma boa ideia devido à complexidade que apresentam (PRATHER et al., 2019). Como consequência, muitos aprendizes acabam por não entender o problema adequadamente e perdem um tempo considerável na criação de algoritmos incorretos. Em situações mais graves, não conseguem progredir, ainda que saibam os conceitos de programação.

Assim, dedique um tempo para ler o enunciado dos exercícios de programação ou outros materiais que estejam disponíveis, para avaliar se compreendeu os conceitos que são apresentados e se preciso faça uma releitura.

Caracterizar o problema

Uma parte fundamental da programação é a construção de uma visão sólida sobre o problema. Para isso é preciso identificar seus principais conceitos e construir um entendimento adequado sobre eles.

Uma estratégia eficaz para compreender os detalhes de um problema é dividi-lo em partes menores (GUARDA et al., 2020). Vamos ilustrar com o exemplo anterior e detalhar como esse

problema pode ser dividido em subproblemas. Para facilitar vamos repetir a escrita deste enunciado aqui:

> Leia quatro números do teclado que nomearemos como, A, B, C e D, respectivamente. A seguir, calcule o produto de A e B, chamaremos de X; e o produto de C e D, que chamaremos de Y. Por fim, subtraia X - Y e apresente o resultado no monitor do usuário.

Observe que será preciso ler quatro números do teclado (subproblema 1), seguido pelo cálculo do produto entre eles, em que os resultados são nomeados de X e Y (subproblema 2). Na sequência realizar a subtração de X e Y (subproblema 3), e apresentar o resultado para o usuário da aplicação (subproblema 4).

Identificar o conjunto de subproblemas ajuda na formação do entendimento lógico sobre o que precisará ser resolvido. Com essa organização, um programador pode escrever os códigos de programação para cada um dos subproblemas e ao término terá resolvido o problema maior.

Também é possível identificar a relação entre eles, por exemplo, o subproblema 2 depende do subproblema 1, uma vez que somente é possível realizar um cálculo após ler os números do teclado.

Por fim, também é importante associar o problema em questão a outros que você já resolveu. Tente lembrar outros exercícios ou projetos que parecem similares, pois em muitos casos parte das soluções pode ser reaproveitada, como também ajudará a clarear seu entendimento sobre o problema.

Representar o problema

Ao chegar a esta etapa você já possui um entendimento (ainda que parcial) sobre o problema e dos detalhes que o compõem. Por isso,

para problemas mais complexos pode ser preciso organizar essas ideias.

O uso de gráficos é uma técnica reconhecida para representação e organização de informações. Existem diferentes formatos que podem ser adotados, sendo o *fluxograma* uma das estratégias mais comuns para criação de códigos iniciais de programação (DE JESUS, 2011). Esse tipo de gráfico representa uma sequência de ações/etapas que possuem alguma relação entre si. Por exemplo, os subproblemas que identificamos na etapa anterior poderiam ser representados juntamente com a sequência em que deveriam ser resolvidos, como ilustrado na figura a seguir.

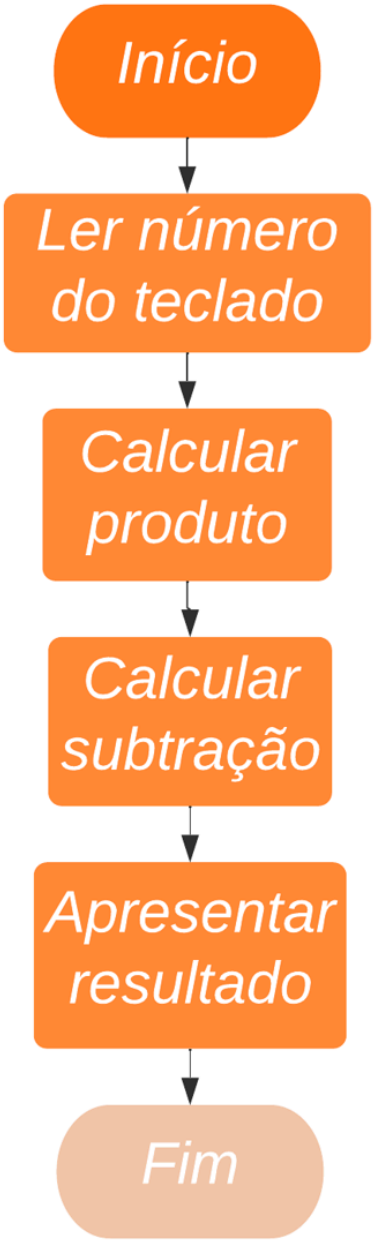

Figura 2.1: Fluxograma com a sequência dos subproblemas identificados no enunciado.

O fluxograma é útil por uma série de razões, dentre as quais destacamos a capacidade de identificar a relação de ordem entre os subproblemas, como também possibilitar a visualização do trabalho que precisa ser realizado.

Existem outras formas de representação de um problema, como escrever a sua lógica de resolução em Português. Essa estratégia é útil, pois o processo de criação de um código demanda um esforço cognitivo para a definição dos recursos de programação que serão utilizados, lembrança da sintaxe, entre outros (WANG e CHIEW, 2010). Como o objetivo dessa etapa é elaborar um melhor entendimento sobre o problema e desenhar uma solução, realizar as ações descritas neste momento em uma linguagem de programação pode atrapalhar.

Assim, a proposta é que após estabelecer um entendimento do problema, você elabore a lógica do programa em em português. No exemplo a seguir apresentamos um trecho de código escrito em português para controlar o saque nos caixas de autoatendimento bancário:

1. Cliente escolheu a opção de saque.
2. Requisitar o valor;
3. Verificar se há saldo na conta, comparando o seu valor com o solicitado no saque;
4. Se houver, subtrair o valor sacado do saldo bancário; liberar o dinheiro solicitado e apresentar uma mensagem: “Saque realizado com sucesso”;
5. caso não exista saldo, então deve apresentar uma mensagem: “Saldo insuficiente.”

Observe que o texto acima está em uma linguagem compreensível para a maioria das pessoas. Além de ajudá-lo(a) no desenho da solução, caso existam outras pessoas envolvidas você poderá conversar com elas para validar se suas suposições estão corretas.

Resolver o problema

Nesta etapa vamos utilizar os recursos de programação para criar uma solução para o problema apresentado. Tudo o que foi planejado nas etapas anteriores servirá de base para as nossas ações durante a escrita do código de programação.

É importante ter definido o que precisa ser feito (esboçado nas seções anteriores) e também pensar em como deve ser implementado. Nesse sentido, devemos pensar em quais recursos da programação serão utilizados. No exemplo do caixa bancário, deve-se definir o valor de saque e saldo do cliente serão representados; o que será preciso para construir uma interface gráfica para pedir e apresentar informações ao cliente; qual recurso será utilizado para verificar se o saque será realizado. O conhecimento que a programadora possui sobre a linguagem de programação é fundamental para o seu sucesso.

Alguns erros podem surgir durante a escrita do código, seja pela falta de um entendimento correto sobre o problema ou por não saber utilizar os recursos de programação corretamente. É fundamental ser resiliente nesta etapa, pois as dificuldades podem ser um pouco desmotivantes. O importante é entender que você está em uma fase de aprendizagem e não se cobrar além dos seus limites. Também é importante reconhecer onde está errando e encontrar meios para superá-los - ler este livro é um caminho :). Por fim, a prática é fundamental ao aprendizado de programação e o levará a desenvolver uma experiência que ajudará a superar os desafios ao longo do caminho.

Refletir sobre a solução

O último passo, e um dos mais esquecidos, ocorre ao final da criação dos programas. É preciso refletir sobre a solução que foi criada e também sobre o caminho que o levou até ela. Subdividimos esta ação em duas, por entender que envolvem processos reflexivos diferentes. No primeiro caso (avaliar a solução) pensamos no algoritmo, como melhorá-lo ou mesmo torná-lo mais eficiente (fazê-lo executar mais rapidamente); no segundo caso, estamos preocupados em aperfeiçoar o processo, ou seja, as ações que realizamos até chegar ao produto final - entender o problema, escrita do código, entre outros.

A reflexão passa por uma autoavaliação que deve acontecer do planejamento à execução. Pensamos no que deu certo e o que pode ser melhorado. Por exemplo, um aprendiz pode considerar que teve dificuldades em desenhar um fluxograma e buscar aperfeiçoar isso em uma próxima atividade.

Além dos elementos descritos anteriormente, as ações apresentadas neste capítulo não devem ser compreendidas como um processo rígido e burocrático. Você pode realizar apenas parte delas e apenas quando julgar necessário. Por exemplo, a depender da complexidade do problema você pode não precisar representar o problema.

O que foi apresentado deve ser entendido como parte de um ciclo, onde as ações são realizadas em uma sequência e retornam ao ponto inicial e base para tudo: o entendimento do problema. Também tenha atenção para o fato de que, em muitos casos, será preciso realizar o ciclo múltiplas vezes para conseguir construir uma solução adequada.

Na prática, muitas das ações descritas já são realizadas por você e o propósito deste capítulo foi tornar explícito procedimentos que são reconhecidos como importantes na criação dos algoritmos. Assim, esperamos que você consiga compreender melhor como e quando realizá-los.

Por fim, é fundamental que você busque uma melhoria contínua na realização dessas ações, de forma que o conhecimento adquirido em um ciclo vai alimentar a realização do próximo. Portanto, as lições aprendidas por você devem ser utilizadas para evitar cometer os mesmos erros e repetir as ações que deram um bom resultado.

2.2 Resumo e mapa mental

- Os problemas são uma parte fundamental da programação e saber como entendê-los é fundamental para a criação de

algoritmos corretos;

- Neste capítulo aprendemos quais ações podem ser realizadas para melhorar o processo de resolução de problemas;
- São elas: entender o problema, caracterizá-lo, representá-lo, resolvê-lo e refletir sobre ele. Para cada uma dessas ações foram descritas estratégias que vão ajudar em sua execução.

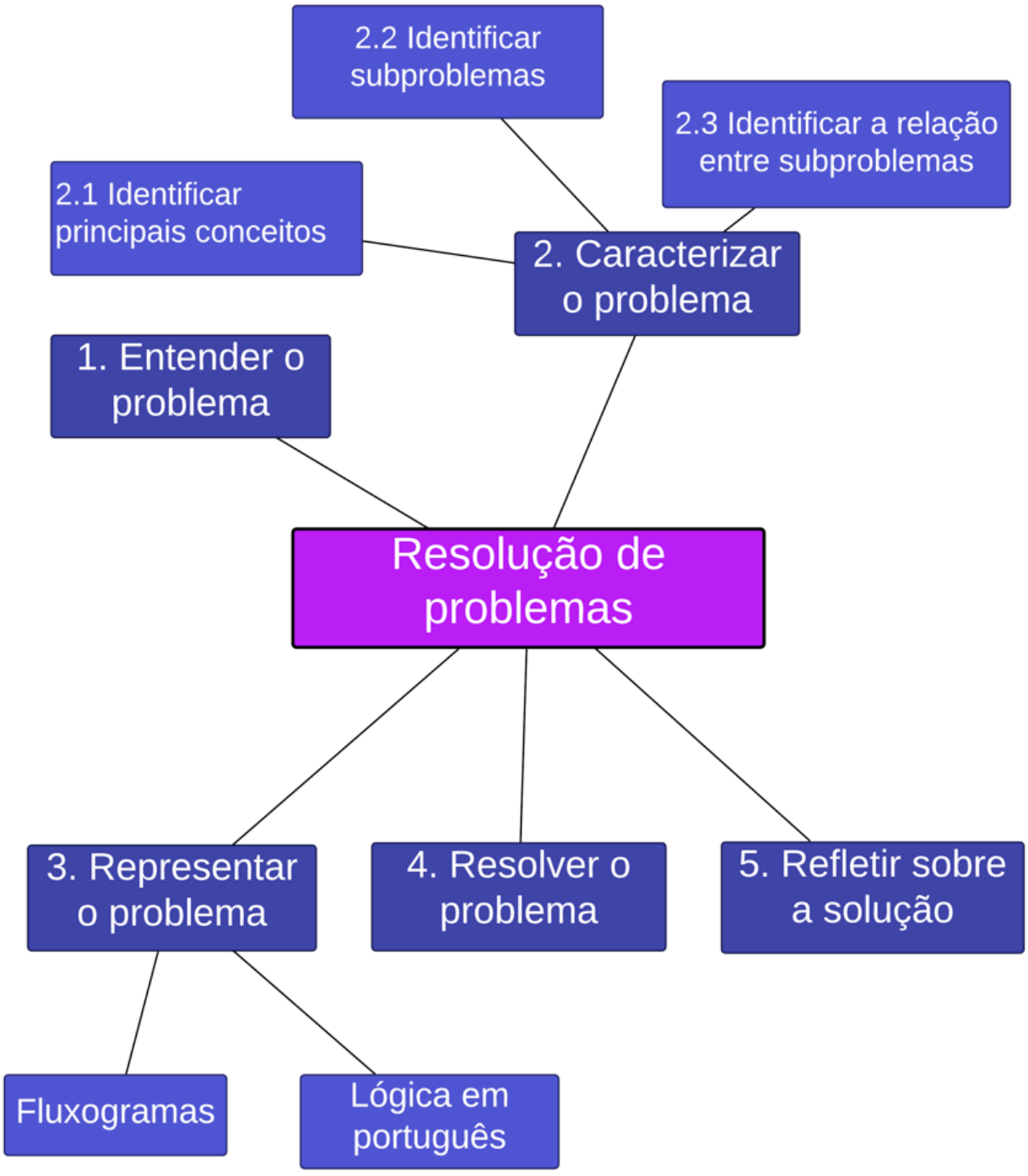
2.2 Identificar subproblemas
2.3 Identificar a relação entre subproblemas
2.1 Identificar principais conceitos
2. Caracterizar o problema
1. Entender o problema
Resolução de problemas
3. Representar o problema
4. Resolver o problema
5. Refletir sobre a solução
Fluxogramas
Lógica em português

Figura 2.2: Mapa mental do capítulo 2.

Capítulo 3
Armazenando dados em nossos programas

1. O que você já sabe?

Um programa, em geral, representa uma solução para um problema. Assim, em sua estrutura, ele representa os elementos que fazem parte deste problema. Por exemplo, um aplicativo de caixa bancário possui uma representação dos dados bancários dos clientes, valores de depósito, saque, entre outros dados que são relevantes para que o problema possa ser resolvido.

2. O que veremos?

Os dados de um programa são uma parte fundamental da sua existência. Para manipulá-los (criar e utilizar) nós utilizamos um conceito conhecido por variáveis. Há uma teoria por trás deste conceito que representa a base de toda a programação, pois os demais conteúdos apresentados dependerão das variáveis. Neste capítulo apresentaremos este importante recurso da programação.

3. Por que é importante?

Todos os programas e jogos eletrônicos dependem de certos dados para funcionar. Assim, as variáveis estarão presentes em todos os softwares e são fundamentais para os próximos assuntos, como condições, repetições e funções. Por isso, tenha muita atenção e veja como as variáveis estão presentes ao seu redor! :)

O que você vai aprender

Ao término deste capítulo você compreenderá:

1. A importância dos dados para os programas;
2. A relação entre dados e variáveis;
3. As partes de uma variável.

3.1 Os dados em nosso dia a dia

Vamos fazer algumas perguntas e temos certeza de que saberá respondê-las: qual o seu nome? Sua idade? O nome da rua onde mora? O símbolo que representa uma operação de soma na matemática? Fácil, não é?!

Você soube responder, pois foi capaz de guardar cada um desses dados em sua memória. Ao mesmo tempo, também foi capaz de recuperá-los quando necessário (quando perguntamos o seu nome, seu cérebro buscou esse dado em sua memória).

Assim como nós, os programas e jogos também precisam guardar dados. Por exemplo, o *Fortnite* armazena a quantidade de munições da sua arma e uma calculadora armazena os números de uma operação matemática.

Os dados utilizados por um programa são armazenados em um conceito conhecido por **variáveis**. Elas estão presentes em todos os programas, como apresentado na tabela a seguir.

Aplicativo	**Exemplos de variáveis**
WhatsApp	Texto digitado, nome do contato e texto de recado.
Fortnite	Quantidade de munições, armas do inventário e quantidade de vida.
Word	texto digitado, cor do texto e tipo de fonte

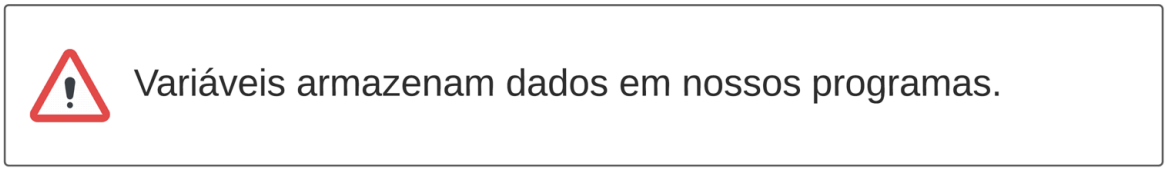

Figura 3.1: Variáveis armazenam dados em nossos programas.

No capítulo anterior quando falamos sobre como programar é resolver problemas, apresentamos um estudo de caso sobre a realização de saques em caixas bancários e esboçamos uma solução. É a partir deste processo (entender o problema e desenho da solução) que identificamos as variáveis que serão necessárias para o software. Precisamos pensar nos dados que vão transitar em nossa aplicação, seja pela entrada do usuário (o valor solicitado para saque), seja por uma saída resultante do seu processamento (o saldo bancário após o saque).

Uma variável será criada para cada dado que um programa precisa armazenar. Para compreender melhor a relação entre variáveis e dados, vamos analisar um exemplo de um aplicativo de edição de textos, como o Word. Existem inúmeros dados que fazem parte do seu funcionamento, como o tipo de fonte que o usuário escolheu, seu tamanho, o texto que o usuário digitou, entre outras informações. Cada um desses dados será guardado em uma variável, como pode visualizar na tabela a seguir:

Variável	Dado
Tipo de fonte	Arial
Tamanho	12

Durante o uso do Word você pode alterar os valores desses dados. Por exemplo, modificar o tipo de fonte para Comic Sans ou o tamanho do texto para 20. Essa possibilidade de modificação dos

dados é que faz as variáveis receberem esse nome. Basta olhar no dicionário Oxford Languages qual o significado da palavra variável:

> "sujeito a variações ou mudanças; que pode variar;"

Portanto, quando pensamos em uma variável estamos falando em guardar um dado que poderá ser alterado no futuro.

3.2 Estrutura de uma variável

Uma variável possui uma estrutura e é preciso compreendê-la antes de partir para a sua criação no Python. Para explicar cada uma de suas partes vamos voltar à analogia com a nossa própria memória, uma vez que as variáveis possuem papel similar, que é guardar dados.

Psicólogos argumentam que o nosso cérebro utiliza um sofisticado mecanismo para gerenciar o grande volume de dados que encontramos durante nossa vida. "Etiquetas" são criadas para auxiliar na memorização e assim facilitar a recuperação de um dado (NORMAN, 2013). Isso ocorre, por exemplo, quando uma pessoa associa que a placa do seu carro é XXZ-1234 ou que a senha do cartão de banco é 123456.

Etiqueta	Dado armazenado
Placa do carro	XXZ-1234
Senha do cartão	123456

Na programação também precisamos de uma forma de etiquetar os dados que serão armazenados para facilitar sua localização. Não é difícil imaginar que, com dezenas ou milhares de variáveis em um programa, as coisas possam sair de controle se você não as

identificar corretamente. Por exemplo, como lembrar em qual variável você guardou a informação sobre o tamanho de fonte que o usuário escolheu? É por isso que as variáveis possuem, além de um dado, um nome (etiqueta) para identificá-las. Vamos ver exemplos de variáveis do Word e como poderiam ser seus nomes e dados:

Nome da variável (etiqueta)	Valor (dado)
`tipo_fonte`	Arial
`tamanho_fonte`	11
`texto`	Olá, este é um texto

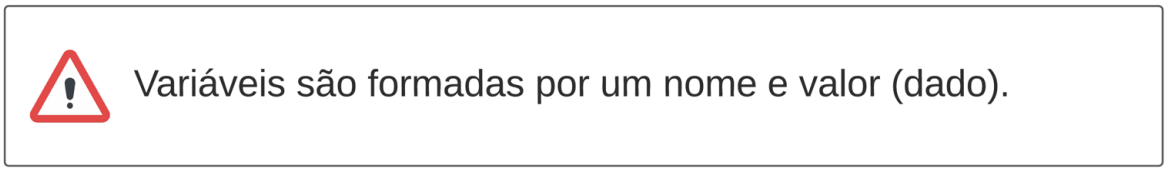

Figura 3.2: Variáveis são formadas por um nome e valor (dado).

Quem define o nome da variável é a pessoa programadora, que pode nomeá-las como quiser. No entanto, existem algumas regras e recomendações que precisam ser seguidas. Por exemplo, não podemos usar espaço e devemos evitar sempre que possível a acentuação. Vamos conhecer as restrições que se aplicam:

Restrição	Nome incorreto	Nome correto
Não deve ter acentos *	`pontuação`	`pontuacao`
Caracteres especiais, como `@$%^` , entre outros *	`total_dinheiro$`	`total_dinheiro`
Começar com números	`3nota`	`terceira_nota`

Restrição	Nome incorreto	Nome correto
Ter espaço	`nome usuario`	`nome_usuario`

* *A partir da versão 3 do Python é possível utilizar esse recurso, porém não é muito comum fazer isso.*

Além dessas restrições, existem boas práticas na hora de nomear suas variáveis. Primeiramente devemos dar nomes que representem o dado que a variável armazena. Vamos ver um mau exemplo: suponha que uma variável de nome `s` foi criada. O que ela armazena? Essa letra pode ter muitos significados: para você, pode significar que a variável guarda uma senha; para mim, pode significar que ela vai armazenar o valor do salário de um funcionário. Como podemos perceber, não é um bom nome para variável, pois pode causar confusão.

Assim, devemos utilizar nomes que indicam o que a variável representa. Por exemplo, o nome `pontuacao` é adequado para armazenar a pontuação do jogador em um jogo (lembra que devemos evitar acentos gráficos?). Para armazenar o telefone de uma pessoa podemos nomear de `telefone` ou `numero_telefone` (não podemos ter espaços!). Armazenar o login de um usuário na variável `login_usuario`.

Perceba que algumas variáveis foram formadas por duas palavras. Como não podemos ter espaço, uma prática recomendada pelo Python é omitir preposições e artigos, como também separar as palavras com `_` (subtraço). Isso não é obrigatório, mas tornou-se uma boa prática de desenvolvimento.

Por fim, tenha atenção pois o Python diferencia palavras maiúsculas de minúsculas nos nomes das variáveis. Por exemplo, os nomes de variáveis: `totalfotos`, `TOTAL_FOTOS`, `total_Fotos` e `Total_Fotos`, representam quatro variáveis diferentes.

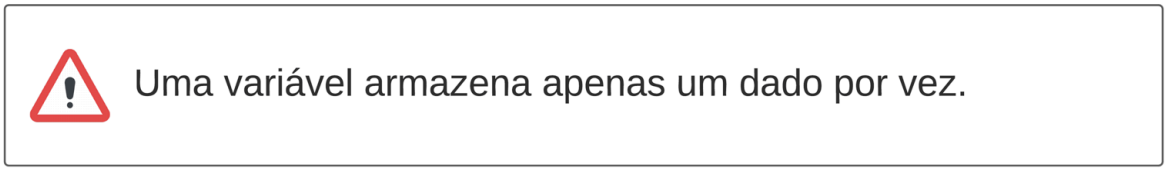

Figura 3.3: Uma variável armazena apenas um dado por vez.

Questões

1) O que são variáveis na programação?

2) Por que as variáveis recebem esse nome?

3) O que aconteceria se não fosse possível armazenar dados em um programa?

4) Assinale com V (verdadeiro) ou F (falso) as afirmações a seguir:

```
( ) O nome de uma variável pode começar com número.

( ) Podemos utilizar espaços no nome de uma variável. Assim, a variável de
nome "senha do usuário" é considerada válida.

( ) Podemos utilizar acentos e caracteres especiais nos nomes das
variáveis, porém devemos evitá-los.

( ) Uma variável pode armazenar mais de um valor.
```

5) Quais nomes de variáveis estão INADEQUADOS?

```
a) total_inimigos_derrotados
b) total_dinheiro$$
c) %_de_acertos
d) quantidade de mensagens
```

6) Imagine que você deseja armazenar o nome de uma pessoa em uma variável, qual seria um bom nome para ela?

a) n
b) senha
c) nome

7) Por que as variáveis precisam de um nome?

Respostas

1) O que são variáveis na programação?

R: Uma variável representa um dado necessário ao funcionamento do programa.

2) Por que as variáveis recebem esse nome?

R: O dado armazenado em uma variável pode ser alterado. Por exemplo, imagine o seu status de relacionamento no Facebook, você pode alterá-lo para refletir as diferentes fases de sua vida (solteiro, casado, entre outros). Essa possibilidade de modificar o dado que armazena é que faz com que as variáveis recebam este nome.

3) O que aconteceria se não fosse possível armazenar dados em um programa?

R: Dificilmente existiriam programas sem variáveis. Como enviar uma mensagem no WhatsApp para um amigo se o programa não armazenou seu número de telefone? Ou se não guardou o texto que escreveu?

4) Assinale com V (verdadeiro) ou F (falso) as afirmações a seguir:

(F) O nome de uma variável pode começar com número.

(F) Podemos utilizar espaços no nome de uma variável. Assim, a variável de nome "senha do usuário" é considerada válida.

(V) Podemos utilizar acentos e caracteres especiais nos nomes das variáveis, porém devemos evitá-los.

(F) Uma variável pode armazenar mais de um valor

5) Quais nomes de variáveis estão INADEQUADOS?

```
a) total_inimigos_derrotados
b) total_dinheiro$$
c) %_de_acertos
d) quantidade de mensagens
```

R: Se você respondeu que as três últimas variáveis (b, c e d) estão com nomes inadequados, acertou. Duas delas possuem símbolos especiais (`$$` e `%`) e a última possui espaço em seu nome.

6) Imagine que você deseja armazenar o nome de uma pessoa em uma variável, qual seria um bom nome para ela?

```
a) n
b) senha
c) nome
```

R: Devemos evitar nomes que não representam a informação que queremos armazenar na variável. No exemplo a), a letra `n` pode ter inúmeros significados, por isso não devemos utilizá-la nesta situação. No exemplo b), não é interessante utilizar a palavra senha, pois o que queremos armazenar é um nome. Logo, a opção correta é a letra c).

7) Por que as variáveis precisam de um nome?

R: Por meio do nome da variável é que conseguimos guardar e recuperar o dado armazenado.

3.3 Resumo e mapa mental

- Neste capítulo aprendemos que os programas manipulam (guardam e recuperam) dados para funcionarem corretamente;
- Vimos que os dados em um programa são guardados em variáveis;

- As variáveis recebem esse nome, pois permitem que seus dados sejam alterados;
- Por fim, vimos que as variáveis são formadas por um nome e um dado que armazenam.

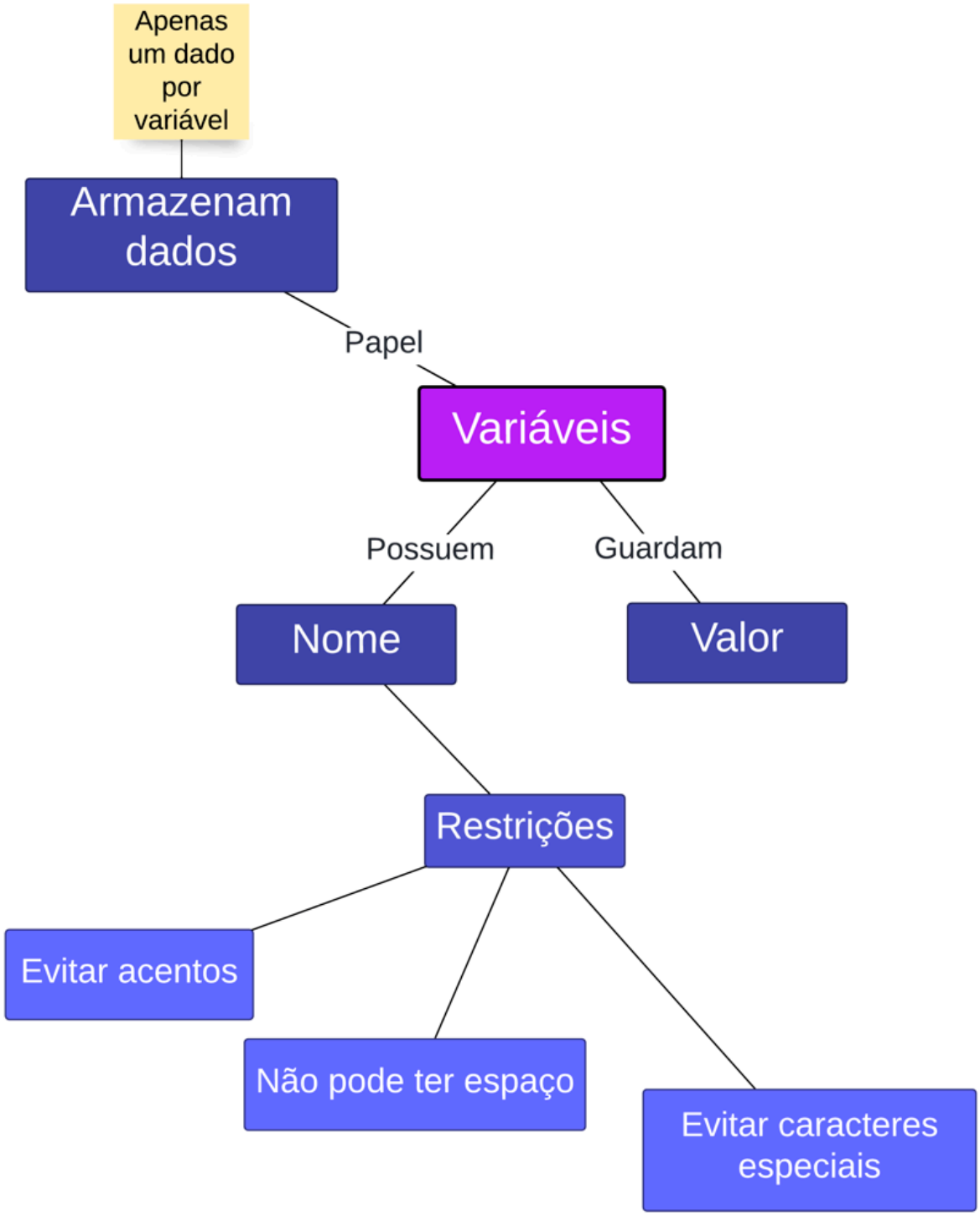
Apenas um dado por variável
Armazenam dados
Papel
Variáveis
Possuem
Guardam
Nome
Valor
Restrições
Evitar acentos
Não pode ter espaço
Evitar caracteres especiais

Figura 3.4: Mapa mental do capítulo 3.

Capítulo 4
Criação de variáveis

1. O que você já sabe?

Ao longo do seu funcionamento, os programas utilizam inúmeros dados que são guardados em variáveis. Esses recursos são representados na programação por meio de um nome e do dado que armazenam.

2. O que veremos?

Até o momento, apresentamos a teoria sobre variáveis e neste capítulo mostraremos na prática como realizar a sua criação e uso com o Python. Você também conhecerá muitas das funcionalidades que podem ser realizadas com as variáveis.

3. Por que é importante?

Ao longo do livro e da sua carreira na programação você criará milhares de variáveis, que são a base de um software. Assim, é fundamental entender a sintaxe utilizada pelo Python para realizar a criação e uso desse recurso.

O que você vai aprender

Ao término deste capítulo você compreenderá:

1. A sintaxe para criação e uso das variáveis no Python;
2. Os tipos de dados que podem ser armazenados em variáveis;
3. Como realizar operações matemáticas e textuais nas variáveis.

4.1 Minha primeira variável

Agora que sabemos a teoria por trás das variáveis, vamos aprender como criá-las no Python. Antes, precisamos lembrar de duas informações importantes: a) uma variável possui um nome e b) ela armazenará um dado. Logo você perceberá a relação deste conhecimento com a sintaxe de uma variável.

A seguir, ilustramos uma variável criada para armazenar o tamanho de uma fonte em um editor de textos:

```
tamanho_fonte = 10
```

Caso ainda não saiba, isso é um código de programação. Dá para imaginar que os programas e jogos eletrônicos que você utiliza são formados por instruções como essa?!

O código acima possui apenas uma instrução que deve ser lida da seguinte forma: **a variável de nome** `tamanho_fonte` **guarda o dado 10**. Veja a figura a seguir.

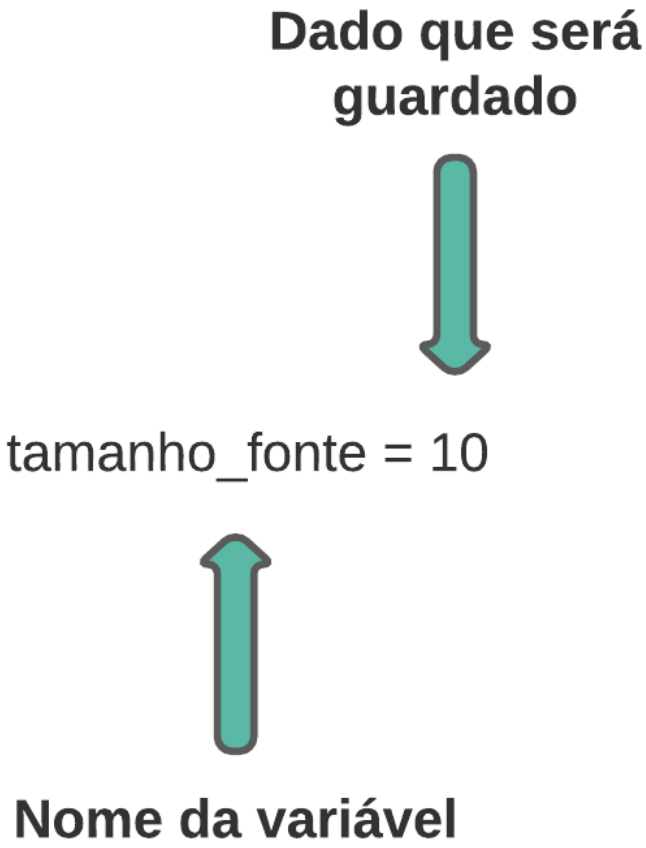

Figura 4.1: Sintaxe de uma variável.

Precisamos respeitar esse formato, uma regra conhecida na programação por **sintaxe**. É dessa forma que o computador é capaz de compreender que você deseja criar uma variável e armazenar um dado nela. Para explicar a sintaxe da declaração da variável `tamanho_fonte`, vamos quebrá-la em três partes:

Parte	Descrição
`tamanho_fonte`	É a primeira informação que devemos escrever e indica qual será o nome que

Parte	Descrição
	daremos para nossa variável.
sinal de = (igualdade)	Também conhecido por **operador de atribuição**, é usado para separar o nome do dado que será guardado.
10	Representa o dado que queremos armazenar na variável.

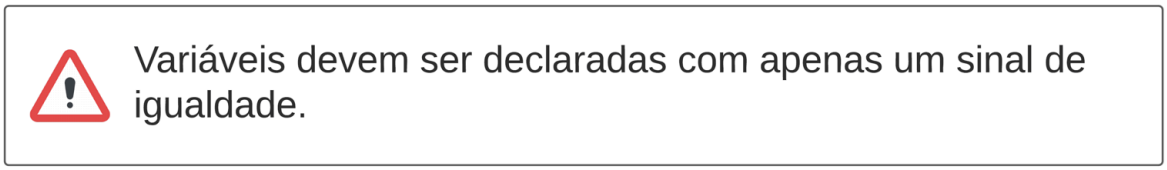

Figura 4.2: Variáveis devem ser declaradas com apenas um sinal de igualdade.

4.2 Modificando o dado

Como falamos no capítulo anterior, as variáveis recebem esse nome porque permitem modificar (variar) o dado guardado. No exemplo a seguir, demonstramos como realizar essa operação:

```
tamanho_fonte = 10
tamanho_fonte = 11
```

Você sabe qual dado a variável `tamanho_fonte` vai armazenar? Primeiramente, observe que esse algoritmo é um pouco diferente do anterior, pois possui duas linhas. Na prática seus algoritmos terão muitas linhas (em alguns casos, milhares, quando eles forem mais complexos). Assim, para responder à pergunta é preciso lembrar como o computador lê o nosso algoritmo, que é parecido como lemos um livro: inicia na primeira linha, segue da esquerda para a

direita até o seu final. Quando isso ocorre, ele avança para a linha seguinte e repete o procedimento até não existirem mais linhas.

Voltando à pergunta, o computador executará a primeira linha e identificará que precisa criar uma variável e guardar o dado 10. Ao executar a segunda linha, ele sabe que já existe uma variável chamada `tamanho_fonte` e por isso modificará o seu dado para 11. Esse exemplo ilustra uma outra característica da criação de programas: **a execução de uma linha do nosso código será afetada pelo que ocorreu nas linhas anteriores**.

4.3 Recuperando dados

Além de armazenar dados, outra característica importante das variáveis é que elas também permitem recuperar o que foi guardado. Veja o exemplo a seguir:

```
tamanho_fonte = 10
outra_variavel = tamanho_fonte
```

Na primeira linha desse algoritmo, criamos uma variável `tamanho_fonte` e armazenamos o dado 10. Na segunda linha, criamos uma variável nomeada `outra_variavel`, que armazenará o dado contido na variável anterior. Essa linha é importante, pois demonstra que o acesso ao dado de uma variável é feito por meio do seu nome. Além disso, percebe-se que é possível recuperar essa informação e em seguida guardá-la em outra variável. Ao término da execução desse código teremos duas variáveis que armazenam o número 10, cada uma.

Aqui vem um detalhe importante, pois, apesar de as duas variáveis armazenarem o dado 10, elas são independentes, ou seja, se futuramente você modificar o valor de uma, isso não afetará a outra. Observe:

```
tamanho_fonte = 10
outra_variavel = tamanho_fonte
tamanho_fonte = 20
```

O que falamos anteriormente é exemplificado com esse algoritmo. Ao término de sua execução, a variável `tamanho_fonte` terá como valor 20 e a variável `outra_variavel` terá o valor 10.

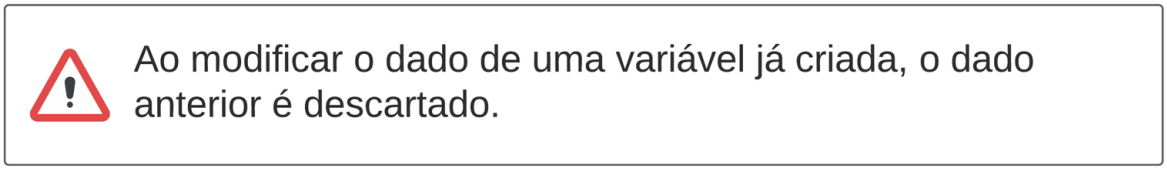

Figura 4.3: Ao modificar o dado de uma variável já criada, o dado anterior é descartado.

Vamos apresentar a seguir um problema comum durante o uso de variáveis. Observe o código, você consegue identificar o que está errado com ele?

```
tamanho_fonte = novo_tamanho
novo_tamanho = 20
```

Explicaremos, na linha 1 estamos criando uma variável `tamanho_fonte` para armazenar o dado presente na variável `novo_tamanho` . O problema é que neste momento, essa variável ainda não existe, pois ela somente é criada na linha 2. Assim, o Python acusará um erro do tipo **NameError**, que ocorre, dentre outros motivos, quando tenta-se utilizar uma variável inexistente.

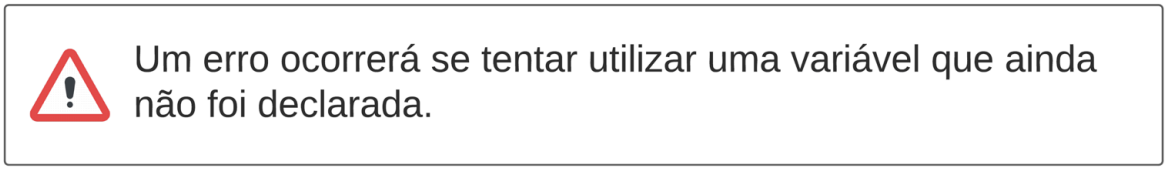

Figura 4.4: Um erro ocorrerá se tentar utilizar uma variável que ainda não foi declarada.

Erros fazem parte da criação de programas e acontecem com iniciantes e profissionais. Criamos o capítulo 5 com a finalidade de falar sobre eles, pois não há como avançar na programação sem saber como lidar com esse problema.

4.4 Variáveis numéricas

Já vimos que as variáveis podem guardar valores numéricos. Na hora de representar alguns números é preciso considerar as diferenças que existem da nossa matemática tradicional. Por exemplo, em números na casa dos milhares, milhões, ou mais, não utilizamos pontos e vírgulas. Veja o algoritmo a seguir, onde vamos declarar uma variável para armazenar o valor 1.000 (mil):

```
saldo_bancario = 1000
```

Observe a ausência de ponto entre o número 1 e 0. É preciso compreender que para a programação os valores 1000 e 1.000 representam dados diferentes. O primeiro representa o numeral 1000, enquanto o segundo representa o número 1, seguido de três casas decimais de valor zero.

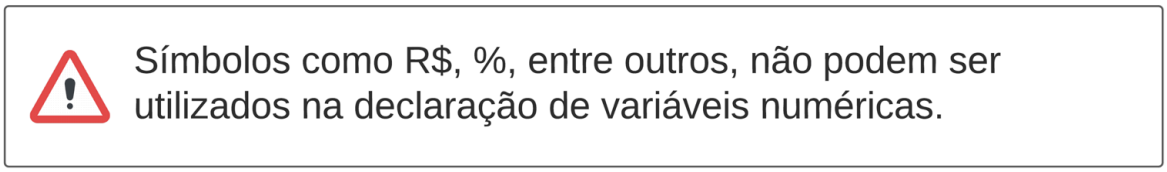

Figura 4.5: Símbolos como R$, %, entre outros, não podem ser utilizados na declaração de variáveis numéricas.

Uma outra diferença ocorre na representação dos números com casas decimais, que são conhecidos na programação por "flutuantes" (do inglês *floating*). No Brasil utilizamos , (vírgulas) para separar a parte inteira, mas na programação devemos seguir o

formato americano que utiliza um . (ponto), como ilustrado a seguir:

```
tamanho_fonte = 10.5
```

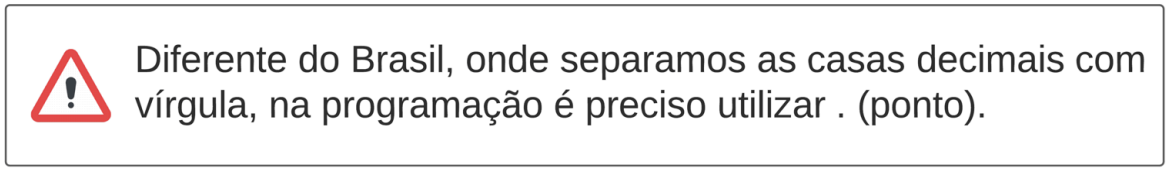

Figura 4.6: Diferente do Brasil, onde separamos as casas decimais com vírgula, na programação é preciso utilizar . (ponto).

A forma como representamos números percentuais também difere. Imagine que desejamos guardar a informação de que o zoom da página do Word está em 70%. Você deve ter pensado no seguinte formato:

```
zoom = 70%
```

No entanto, o Python não utiliza esse símbolo para esta finalidade. Assim, para saber como representar esses números, devemos lembrar de conceitos básicos de matemática:

- 70% é igual a 70/100, que é igual a 0.7
- 8% é igual a 8/100, que é igual a 0.08

Como pode observar, um número fracionário pode ser representado como um número decimal. Assim, o formato correto para representar o dado anterior no Python seria:

```
zoom = 0.7
```

É importante compreender como guardar dados percentuais, pois é muito comum utilizá-los na programação.

O Python também não aceita símbolos diversos na declaração de números. Por exemplo, para indicar que um número representa uma

determinada quantia em dinheiro, não podemos utilizar os símbolos R$, £, entre outros. Devemos ignorá-los:

```
preco_venda = 500
```

Questões

1) A declaração de uma variável é composta por três partes, quais são elas?

2) Assinale V ou F para as afirmações a seguir:

```
( ) A declaração de uma variável segue a sintaxe: nome = valor

( ) A declaração de uma variável segue a sintaxe: nome == valor

( ) A seguinte declaração de variável está correta: dado = 30%

( ) A seguinte declaração de variável está correta: dado = R$ 30

( ) A seguinte declaração de variável está correta: dado = 0.3
```

3) Explique como o computador lê o nosso algoritmo.

4) Declare duas variáveis, uma com número inteiro e outra com um número decimal.

5) Como representar variáveis que guardam os números percentuais a seguir: 30%, 5%, 75% e 6.5%.

6) Qual o problema nos código a seguir:

```
dado = 30.5
outro_dado = mais_um_dado
```

Respostas

1) A declaração de uma variável é composta por três partes, quais são elas?

R: Nome, igualdade e dado.

2) Assinale V ou F para as afirmações a seguir:

```
( V ) A declaração de uma variável segue a sintaxe: nome = valor

( F ) A declaração de uma variável segue a sintaxe: nome == valor

( F ) A seguinte declaração de variável está correta: dado = 30%

( F ) A seguinte declaração de variável está correta: dado = R$ 30

( V ) A seguinte declaração de variável está correta: dado = 0.3
```

3) Explique como o computador lê o nosso algoritmo.

R: Um computador lê as linhas do código de cima para baixo, da direita para a esquerda.

4) Declare duas variáveis, uma com número inteiro e outra com um número decimal.

```
dado = 5
outro_dado = 5.5
```

5) Declare as variáveis que guardam os números percentuais a seguir: 30%, 5%, 75% e 6.5%.

```
dado = 0.3
dado = 0.05
dado = 0.75
dado = 0.065
```

6) Qual o problema nos código a seguir:

```
dado = 30.5
outro_dado = mais_um_dado
```

R: A variável `mais_um_dado` está sendo utilizada na linha dois, mas ainda não foi declarada.

4.5 Variáveis que armazenam palavras/textos

Para nossa felicidade, a vida não é feita somente de números. O seu nome, este livro, a sua letra de música preferida são formados por palavras e é claro que os nossos programas também precisam armazenar informações desse tipo. Já imaginou usar o WhatsApp ou Word sem textos? Essas aplicações simplesmente não existiriam.

Na programação as variáveis que armazenam textos são chamadas de **Strings**. As variáveis deste tipo possuem uma característica em especial, pois os seus valores **devem estar entre `" "` (aspas) que podem ser duplas ou simples**. Vamos ver um exemplo:

```
texto_escrito = "Este livro é legal!"
outro_texto = 'Vou aprender a programar.'
```

Atenção: caso tente declarar uma variável do tipo *String* sem utilizar aspas o Python acusará um erro que pode ser do tipo **NameError** ou **SyntaxError**. Além disso, você não pode misturar os diferentes tipos de aspas, ou seja, se começou com aspas duplas, não pode finalizar com aspas simples.

Há ainda uma sintaxe especial utilizada apenas na criação de textos extensos que irão ocupar múltiplas linhas. Nestes casos nós utilizamos:

```
texto = """
primeira linha
segunda linha
terceira linha
"""
```

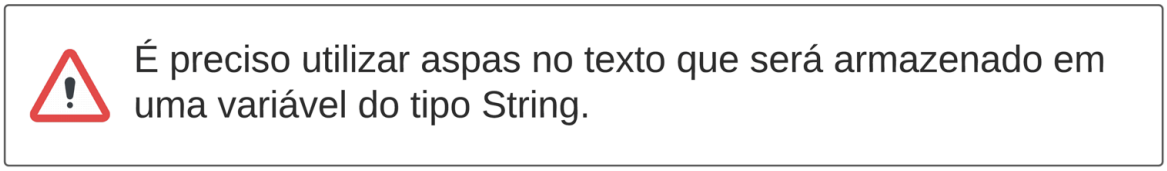

Figura 4.7: É preciso utilizar aspas no texto que será armazenado em uma variável do tipo String.

O procedimento para acessar os dados guardados em variáveis do tipo *String* é o mesmo utilizado com as variáveis numéricas. Vamos referenciar o nome da variável de interesse:

```
texto_escrito = "Este livro é legal!"
outro_texto = texto_escrito
```

Na primeira linha deste código nós criamos uma variável `texto_escrito`. Na segunda linha criamos outra variável nomeada `outro_texto` que vai armazenar o dado contido na variável criada na linha anterior. **Observe que para acessar o dado de uma variável *String* nós não utilizamos aspas, apenas o nome da variável**.

Outra característica das *Strings* é que elas também podem guardar números. Mas atenção, nesses casos são textos e não dados do tipo numérico. Veja o algoritmo:

```
um_texto = "2"
```

A variável `um_texto` não armazena o número dois, mas sim uma *String* cujo valor é um texto *2*. O exemplo apresentado é útil para que você crie consciência sobre os diferentes tipos de dados que existem na programação. Para o computador, o dado *2* é diferente da *String* "2", isso implica que operações matemáticas podem ser feitas no primeiro, mas não no segundo.

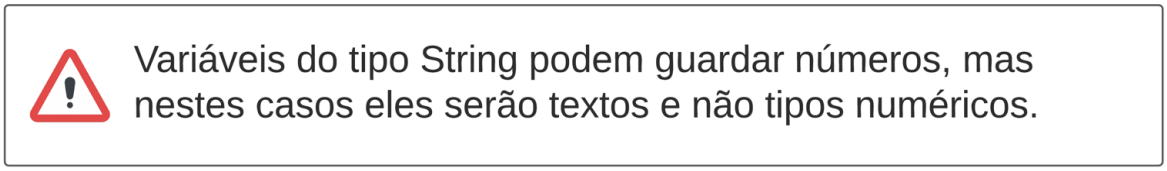

Figura 4.8: Variáveis do tipo String podem guardar números, mas nestes casos eles serão textos e não tipos numéricos.

Podemos realizar operações em *Strings*, como unificar o texto de duas ou mais variáveis. Por exemplo, considere uma variável que guarda o primeiro nome de uma pessoa e outra que armazena o sobrenome:

```
primeiro_nome = "Gabriel"
segundo_nome = "Fortes"
```

Podemos criar uma terceira variável que guardará os dados contidos nas duas variáveis, resultando em um texto único: Gabriel Fortes. O processo de união de variáveis do tipo *String* chama-se **concatenação**. Para realizar esse procedimento nós utilizaremos o operador de concatenação + (sinal de mais). Veja o exemplo a seguir:

```
primeiro_nome = "Gabriel"
segundo_nome = "Fortes"
nome_completo = primeiro_nome + segundo_nome
```

A concatenação é realizada na última linha deste código e o resultado desta ação guardamos na variável `nome_completo`. O seu valor é o texto `Gabriel Fortes`, que foi formado a partir da concatenação das variáveis `primeiro_nome` e `segundo_nome`.

Podemos também concatenar uma variável *Strings* com um texto diretamente:

```
primeiro_nome = "Gabriel"
saudacao = "Olá "+primeiro_nome
```

A concatenação somente pode ser feita entre variáveis do tipo *String*. Se você tentar concatenar diferentes tipos de dados, como um número e uma *String*, ocasionará um erro. O código a seguir representa esse problema:

```
numero = 10
texto = "a"
resultado = numero + texto
```

Ao executar este código o Python acusará um erro do tipo **TypeError**.

Por fim, tentar utilizar o símbolo de aspas dentro de um texto pode ocasionar um problema, veja a seguir:

```
texto "Exemplo com o símbolo de " (aspas)" # Apresenta um erro
```

Observe que o texto apresenta três aspas, a primeira e a última são utilizadas na declaração da *String*. A aspas do meio faz parte do texto. No entanto, o código apresentado ocasionará um erro, pois o computador vai entender que a aspas no meio do texto é utilizada para encerrar as primeiras aspas. Uma confusão, não é?! Assim, quando for preciso escrever o símbolo de aspas dentro de uma *String* temos duas opções. A primeira é utilizar aspas simples na declaração da variável e incluir o símbolo " (aspas duplas) dentro do valor:

```
texto = 'Exemplo com o símbolo de " (aspas)'
```

Neste caso não há conflito, pois o texto está entre aspas simples e a aspas utilizada no meio é dupla.

A segunda forma utiliza o operador de escape `\` (barra invertida) antes do símbolo de aspas. Seu uso possibilita combinar um texto entre aspas duplas e o uso deste tipo de aspas ao mesmo tempo:

```
texto = "Exemplo com o símbolo de \" (aspas)"
```

A barra invertida não aparecerá na saída deste código e resultará em: *Exemplo com o símbolo de " (aspas)*.

Questões

1) O que variáveis do tipo *String* armazenam?

2) Para declarar uma variável *String* devemos colocar o valor entre:

a. Parênteses
b. Colchetes
c. Aspas
d. Asterisco

3) Declare uma variável para representar o valor do seguinte texto: Estou lendo um livro de introdução à programação.

4) Qual o operador utilizado para concatenar textos?

a. igualdade
b. subtração
c. mais
d. aspas

5) Por que utilizamos o operador de concatenação?

6) Realize a concatenação de duas variáveis do tipo texto. Você pode definir os valores que achar mais adequado.

Respostas

1) O que variáveis do tipo *String* armazenam?

R: Variáveis do tipo *String* são usadas para armazenar textos e palavras.

2) Para declarar uma variável *String* devemos colocar o valor entre:

a. Parênteses
b. Colchetes
c. Aspas
d. Asterisco

R: A declaração do valor de variáveis *String* deve ser feita utilizando "" (aspas simples ou dupla).

3) Declare uma variável para representar o valor do seguinte texto: Estou lendo um livro de introdução à programação.

```
texto = "Estou lendo um livro de introdução à programação"
```

4) Qual o operador utilizado para concatenar textos?

```
a. igualdade
b. subtração
c. mais
d. aspas
```

R: O operador de concatenação é o + (mais)

5) Por que utilizamos o operador de concatenação?

R: A concatenação é útil para junção de uma ou mais variáveis *String* em uma única variável.

6) Realize a concatenação de duas variáveis do tipo texto. Você pode definir os valores que achar mais adequado.

```
linguagemProgramacao = "Python"
livro = "Introdução à Programação em "
texto = livro + linguagemProgramacao
```

4.6 Variáveis do tipo booleanas

Nesta seção apresentaremos um tipo especial de dado que você talvez não conheça, mas utilizará com frequência: o tipo booleano. Vamos ilustrar seu uso com um exemplo no WhatsApp, nele, quando você está com acesso à Internet, está conectado; quando não está, seu estado é desconectado. Perceba que nessa situação não existe a situação meio conectado. Imagine agora um aplicativo para

smartphone que permite ligar ou desligar remotamente as lâmpadas da sua casa. Para isso é preciso saber se elas estão acesas ou apagadas. Percebe como nos dois exemplos o dado possui apenas dois estados possíveis (conectado/desconectado e acesa/apagada).

Em situações em que há somente duas possibilidades de valores utilizamos o tipo booleano. Ele é representado pelos dados: `True` (verdadeiro) e `False` (falso), que podem ser usados para representar inúmeras situações. Por exemplo, para indicar que o usuário está conectado à Internet, poderíamos utilizar o valor True e quando desconectado o valor False. A lâmpada acesa pode ser representada por True e apagada por False.

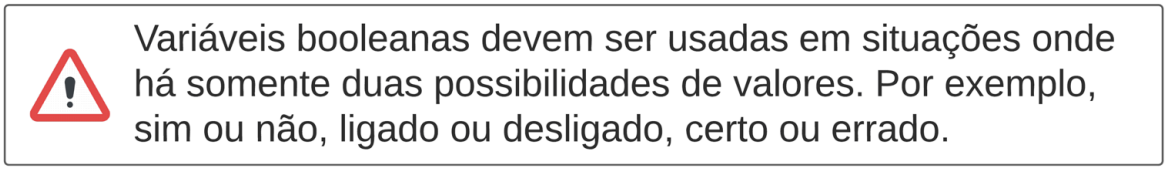

Figura 4.9: Variáveis booleanas devem ser usadas em situações onde há somente duas possibilidades de valores. Por exemplo, sim ou não, ligado ou desligado, certo ou errado.

Vamos ver exemplos de uso de variáveis booleanas no Python:

```
usuario_conectado = True
lampada_acesa = False
```

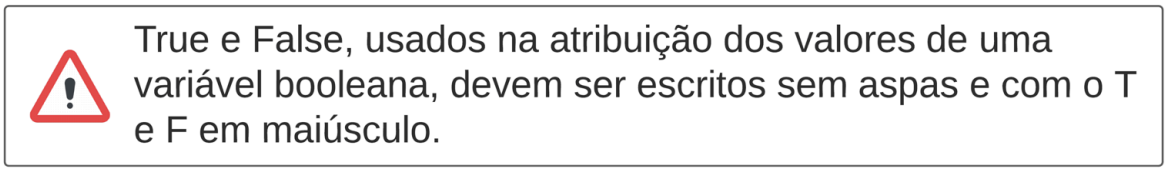

Figura 4.10: True e False, usados na atribuição dos valores de uma variável booleana, devem ser escritos sem aspas e com o T e F em maiúsculo.

Questões

1) Quais são os valores que uma variável booleana pode possuir?

2) Em quais situações nós usamos variáveis booleanas?

3) Marque com V ou F as situações a seguir que podem ser representadas como booleanas:

( ) O estado de uma lâmpada que pode estar acesa ou apagada;

( ) A velocidade de um carro;

( ) O saldo de dinheiro na conta de uma pessoa;

( ) A situação de um estudante que pode estar aprovado ou reprovado.

Respostas

1) Quais são os valores que uma variável booleana pode possuir?

R: True ou False

2) Em quais situações nós usamos variáveis booleanas?

R: Variáveis do tipo booleanas são usadas para representar informações que possuem apenas dois valores possíveis.

3) Marque com V ou F as situações a seguir que podem ser representadas como booleanas:

(V) O estado de uma lâmpada que pode estar acesa ou apagada;

(F) A velocidade de um carro;

(F) O saldo de dinheiro na conta de uma pessoa;

(V) A situação de um estudante que pode estar aprovado ou reprovado.

4.7 Tipos de dados

Percebe-se que o Python possui um conjunto de tipos de dados que nos permite representar diferentes situações: números inteiros, decimais, textos e variáveis booleanas. Ao longo do livro você conhecerá outros tipos, mas para iniciar, são o suficiente.

Neste momento é importante compreender que o dado armazenado em uma variável é quem indica o seu tipo. Consequentemente, ele restringe quais operações podem ser realizadas na variável. Na próxima seção você aprenderá a realizar cálculos matemáticos, sendo que estes somente podem ser realizados em variáveis numéricas. Da mesma forma, a ação de concatenação somente pode ser realizada em variáveis do tipo *String*. A pessoa programadora precisa estar atento ao que vai realizar com uma variável para não violar uma regra da linguagem de programação.

Vamos falar sobre uma outra característica das variáveis em Python que é a capacidade de trocar os tipos de dados que armazenam. Veja no exemplo a seguir:

```
numero = 2
numero = "2"
```

Na primeira linha do código declaramos uma variável *numero* que armazenava o valor numérico dois. Em seguida, trocamos o seu valor para uma *String* com o número dois textual. Esse procedimento ocasionou uma mudança no tipo da nossa variável que iniciou numérica e finaliza como uma *String*.

4.8 Operações matemáticas

Agora que conhecemos alguns dos tipos de dados que o Python permite guardar, vamos falar sobre operações que podem ser

realizadas. Em particular, as ações que fazemos em dados numéricos.

A capacidade de processar operações matemáticas rapidamente é um dos grandes poderes dos computadores. Podemos realizar multiplicações, divisões, entre outras, de forma rápida e livre de erros. Jogos realizam contas com frequência, por exemplo, no jogo Fortnite, cada disparo com a sua arma reduz em um o número de balas, uma operação de subtração.

Vamos aprender como realizar cálculos matemáticos. Continuaremos com o exemplo anterior, iremos representar uma situação onde o jogador tinha 20 balas e após realizar um um disparo perdeu uma. O seguinte algoritmo poderia ser utilizado:

```
quantidade_balas = 20 - 1
```

Há diversas possibilidades de operações matemáticas, mas é preciso compreender as pequenas diferenças nos símbolos que são usados em comparação à matemática tradicional:

```
soma = 3 + 5
subtracao = 3 - 5
multiplicacao = 3 * 5
divisao = 3 / 5
```

Também é possível realizar operações matemáticas diretamente nos valores de variáveis:

```
quantidade_balas = 20
nova_quantidade_balas = quantidade_balas - 1
```

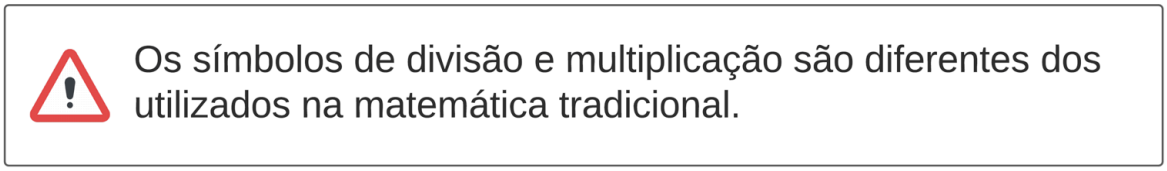

Figura 4.11: Os símbolos de divisão e multiplicação são diferentes dos utilizados na matemática tradicional.

Nesse código, criamos uma variável `quantidade_balas` e guardamos o valor 20; em seguida, declaramos a variável `nova_quantidade_balas`, que terá como valor o dado guardado na variável `quantidade_balas` subtraído de 1, resultando em 19.

Há espaço para melhorar o código anterior, pois utilizamos duas variáveis quando poderíamos ter utilizado apenas uma, veja a seguir:

```
quantidade_balas = 20
quantidade_balas = quantidade_balas - 1
```

Talvez esse código seja um pouco mais difícil de entender do que o anterior, pois na segunda linha estamos atribuindo um valor à variável `quantidade_balas` ao mesmo tempo que utilizamos o valor dessa mesma variável. Para ajudar na compreensão, você deve ler essa instrução da seguinte forma: **o valor da variável `quantidade_balas` será igual ao seu valor atual, que é 20, subtraído 1**. Após a execução dessa linha pelo computador, ele atualizará o valor da variável `quantidade_balas` para 19, e o anterior (20) será descartado.

É possível modificar o dado de uma variável a partir dele mesmo. Dessa forma, utilizamos o seu valor, fazemos uma modificação e o salvamos, sem criar uma nova variável.

Figura 4.12: É possível modificar o dado de uma variável a partir dele mesmo. Dessa forma, utilizamos o seu valor, fazemos uma modificação e o salvamos, sem criar uma nova variável.

Vamos ver outro exemplo um pouquinho mais complexo de mudança de valores nas variáveis. Tente analisar o código a seguir e compreender os valores das variáveis x e y.

```
x = 2
y = 2
x = x + 2
y = x + 2
```

Consegue prever quais serão os valores das variáveis x e y ao término da execução do algoritmo?

Para isso, é preciso entender como a linguagem de programação executa o seu algoritmo. Nas linhas 1 e 2, são criadas duas variáveis: *x* e *y*, cujo valores são 2 e 2, respectivamente. Na linha 3, a instrução `x = x + 2` atualiza o valor da variável *x*, somando o seu valor atual ao 2, resultando em quatro. Por fim, na última linha, o valor da variável *y* é modificado e passa a ser o resultado da operação `x + 2`. Como no momento de execução desta linha a variável *x* já possui o valor de 4, y terá como valor 6 (4 + 2). Parece um pouco confuso e leva um pouco de tempo para se acostumar com essa interpretação do algoritmo, mas não se assuste, a prática é fundamental para se habituar.

Vamos ver um outro exemplo de operação matemática. Imagine que queremos saber a quantidade média de disparos em duas partidas do Fortnite. Para isso, criaremos duas variáveis que guardarão o

total de disparos em cada partida e realizaremos uma divisão. Veja o código adiante. Consegue prever o valor da variável *média*?

```
total_disparos_partida_um = 1000
total_disparos_partida_dois = 500
media = total_disparos_partida_um+total_disparos_partida_dois/2
```

Se você pensou no número 750 errou :). Mas calma, vamos entender o que ocorreu, pois o algoritmo está intencionalmente incorreto. Assim como na matemática, **as operações de multiplicação e divisão precedem as operações de soma e subtração**. Portanto, nesse cálculo, primeiro será realizada a operação de divisão do valor da variável `total_disparos_partida_dois` por 2, e em seguida esse resultado será somado ao valor da variável `total_disparos_partida_um`, o que resultaria em 1250 (1000 + 250).

Obviamente o resultado está errado, pois o cálculo de uma média primeiro deve considerar a soma dos números, antes da divisão. Em situações como essa **é preciso modificar a precedência da operação matemática utilizando () (parênteses)**, assim como na matemática tradicional. Vamos ver como seria o algoritmo correto:

```
total_disparos_partida_um = 1000
total_disparos_partida_dois = 500
media = (total_disparos_partida_um+total_disparos_partida_dois)/2
```

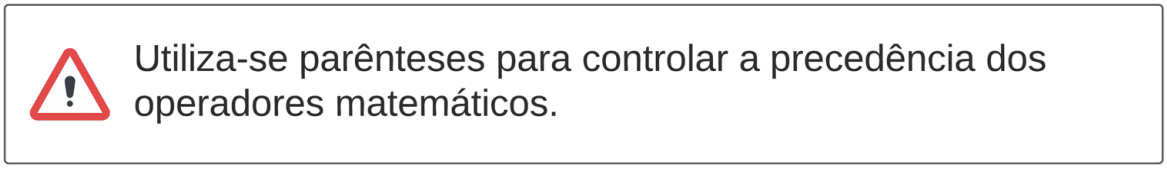

Figura 4.13: Utiliza-se parênteses para controlar a precedência dos operadores matemáticos

Por fim, um último recurso que será útil na hora de manipular suas variáveis numéricas são os operadores de atribuição compostos.

Utilizaremos como exemplo a redução no quantitativo de munições no Fortnite:

```
quantidade_balas = 20
quantidade_balas -= 1
```

A segunda linha do código deve ser lida da seguinte forma: o valor da variável `quantidade_balas` será igual ao seu valor atual (20) menos 1. Poderíamos substituir o valor de 1 (um) por qualquer outro valor ou mesmo fazer uso de uma variável:

```
balas_subtraidas = 3
quantidade_balas = 20
quantidade_balas -= balas_subtraidas
```

As possibilidades de operações compostas são apresentadas na tabela a seguir.

Operação	Operador composto
Soma	`+=`
Subtração	`-=`
Divisão	`/=`
Multiplicação	`*=`

Por fim, um último operador matemático importante é o % (resto de divisão). Apesar de seu símbolo, ele não é utilizado para representar números percentuais. Utilizamos esse operador para descobrir o resto de uma divisão. Considere o exemplo adiante:

```
resto_divisao = 2 % 2
```

Esse algoritmo criará uma variável `resto_divisao` que terá como valor o resto da operação de divisão entre 2 e 2, que é zero. Um dos usos desse operador é para identificar se um número é par ou ímpar. Se o resto da divisão por dois for zero, significa que ele é par.

Questões

1) Escreva os operadores matemáticos utilizados no Python para realizar as seguintes operações matemáticas:

```
( ) Soma
( ) Subtração
( ) Divisão
( ) Multiplicação
```

2) Qual a importância dos operadores matemáticos para a programação?

3) Qual é o valor da variável pontuação após a execução do algoritmo?

```
pontuacao = 10
pontuacao = pontuacao + 3
```

4) Qual será o valor de x após a execução do algoritmo?

```
x = 2
y = x
y = y*2
x = y/2
```

5) O algoritmo a seguir possui um erro, pois não calcula corretamente uma média. O que é necessário para corrigi-lo?

```
pontuacaoUm = 10
pontuacaoDois = 15
pontuacaoTres = 5
media = pontuacaoUm + pontuacaoDois + pontuacaoTres/3
```

6) O que são os operadores de atribuição compostos e quais os benefícios em utilizá-los?

7) Considere um algoritmo com duas variáveis, sendo um número par e outro ímpar. Qual operador poderia ser utilizado para descobrir quem é o par e quem é o ímpar?

Respostas

1) Escreva os operadores matemáticos utilizados no Python para realizar as seguintes operações matemáticas:

(+) Soma (-) Subtração (/) Divisão (*) Multiplicação

2) Qual a importância dos operadores matemáticos para a programação?

R: Com o uso de operadores matemáticos podemos explorar o grande poder computacional para realizar complexos cálculos matemáticos.

3) Qual é o valor da variável pontuação após a execução do algoritmo?

```
pontuacao = 10
pontuacao = pontuacao + 3
```

R: A variável pontuação terá como valor 13.

4) Qual será o valor de x após a execução do algoritmo?

```
x = 2
y = x
y = y*2
x = y/2
```

R: O valor da variável *x* será 2.0.

5) O algoritmo a seguir possui um erro, pois não calcula corretamente uma média. O que é necessário para corrigi-lo?

```
pontuacaoUm = 10
pontuacaoDois = 15
pontuacaoTres = 5
media = pontuacaoUm + pontuacaoDois + pontuacaoTres/3
```

R: É preciso utilizar os parênteses para controlar a precedência das operações de soma, sobre a divisão.

6) O que são os operadores de atribuição compostos e quais os benefícios em utilizá-los?

R: São operadores matemáticos que permitem utilizar o valor de uma variável como base para sua própria modificação. O principal benefício é a simplificação na escrita dessas operações.

7) Considere um algoritmo com duas variáveis, sendo um número par e outro ímpar. Qual operador poderia ser utilizado para descobrir quem é o par e quem é o ímpar?

R: O operador de resto de divisão *%* pode ser utilizado para identificar se um número é par ou ímpar. Basta identificar se o resto da sua divisão por dois é zero, o que significa que ele é par.

4.9 Armazenamento das variáveis no computador

Você pode estar se perguntando onde o computador armazena as variáveis que são criadas ao longo de um programa. Provavelmente, você já ouviu falar de um recurso conhecido por memória RAM (*Random-access Memory*). É nesse componente eletrônico que as variáveis são armazenadas.

As memórias RAM possuem um tamanho (usualmente expresso em gigabytes) que indica a quantidade de informação que conseguem armazenar. Quanto mais memória, mais variáveis podem armazenar, consequentemente mais programas podem ser executados simultaneamente com menor perda de performance (convém lembrar que este não é o único fator que determina isto).

Cada vez que você cria uma variável um pequeno espaço da memória RAM será ocupado por seu valor. Como todos os programas que você utiliza (navegador, editor de texto, jogos, entre outros) são

formados por variáveis, eles também estão consumindo a memória do computador.

A memória RAM possui um mecanismo para controlar as variáveis que estão guardadas. De uma forma simplista, ao criar uma variável, o computador reserva uma pequena porção da memória e associa um endereço a ela. Há uma interação entre o sistema operacional (que gerencia a memória) e a linguagem de programação, para que seja possível recuperar as informações que foram guardadas, e o conceito de endereçamento é fundamental. Para nossa felicidade esse complexo processo é transparente ao programador(a), permitindo o trabalho em um nível mais alto de abstração, onde os dados são gerenciados por meio de nomes, o que torna tudo muito mais fácil.

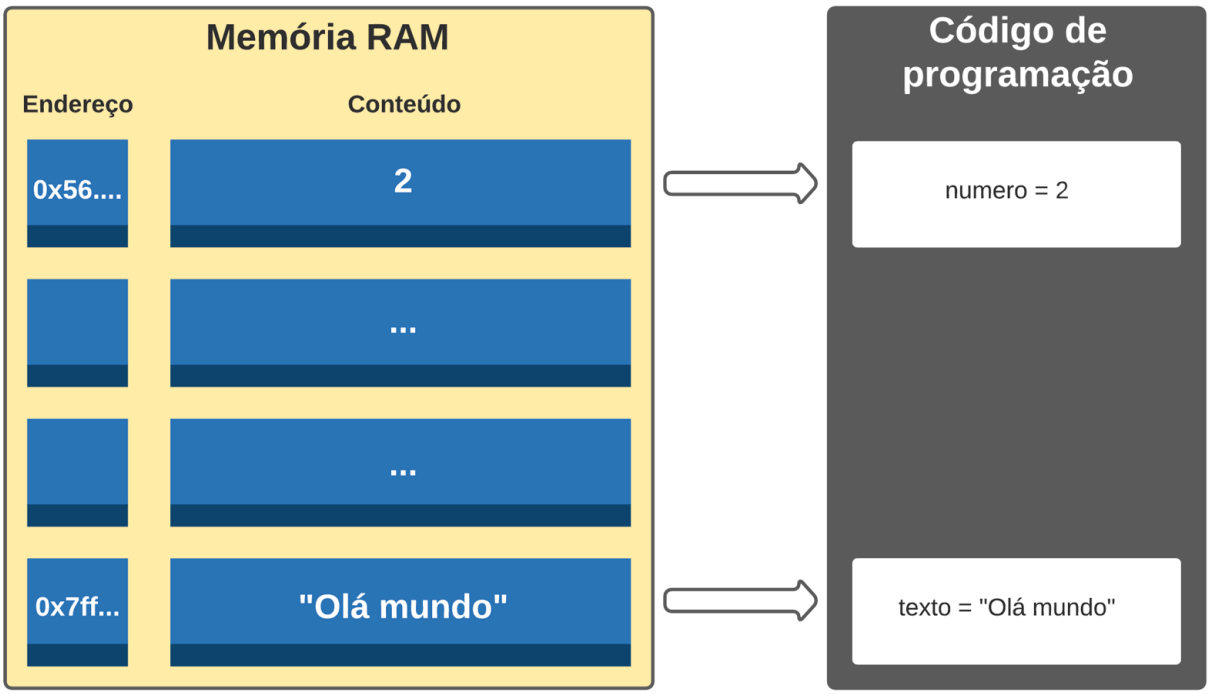

Figura 4.14: Representação simplificada da memória RAM.

Dificilmente você vai se preocupar com o espaço que o seu programa ocupa na memória RAM do computador. Atualmente, o

quantitativo disponível é mais que o suficiente para armazenar todas as suas variáveis e ainda sobrará muito espaço. Para fazer uma analogia, imagine que a memória RAM é como uma imensa piscina olímpica vazia e que cada variável que você criar seria o equivalente a colocar uma pequena gota nesta piscina. Além disso, o sistema operacional gerencia as situações quando a memória é totalmente ocupada.

No entanto, se o seu foco for criar aplicações embarcadas em microcontroladores, como o Arduino, então é preciso otimizar a escrita do seu código de programação, pois a quantidade de memória RAM nestes dispositivos é bastante limitada. Por enquanto não vamos nos preocupar com isto! :)

4.10 Erros mais comuns

Alguns erros são frequentes no uso de variáveis e nesta seção apresentaremos os principais para que você evite cometê-los:

Não utilizar aspas na declaração de uma *String*. Em alguns casos o programador esquece de utilizar aspas.

Código com problema:

```
saudacao = "Olá mundo
```

Código corrigido:

Diferente de outras variáveis, o conteúdo das *Strings* precisa estar entre aspas.

```
saudacao = "Olá mundo"
```

Utilização de variável não declarada. Somente é possível utilizar uma variável após ela ter sido declarada.

Código com problema:

```
saudacao = nova_saudacao
```

Código corrigido:

```
nova_saudacao = "Olá mundo no Python"
saudacao = nova_saudacao
```

Declarar uma variável com duas igualdades. A declaração de uma variável é feita utilizando apenas um sinal de igualdade.

Código com problema:

```
saudacao == "Olá mundo"
```

Código corrigido:

```
saudacao = "Olá mundo"
```

Escrita incorreta das variáveis booleanas. As variáveis booleanas podem assumir dois valores: True ou False, que devem ser escritos desta forma.

Código com problema:

```
variabel_booleana = true
outra_variavel_booleana = "True"
```

Código corrigido:

```
variabel_booleana = True
outra_variavel_booleana = True
```

Utilizar vírgula para separar casas decimais. A separação de casas decimais segue o formato americano onde utiliza-se . (ponto) em vez de vírgula.

Código com problema:

```
salario = 100,5
```

Código corrigido:

```
salario = 100.5
```

4.11 Resumo e mapa mental

- Neste capítulo aprendemos como criar e utilizar variáveis no Python;
- Conhecemos os principais tipos de variáveis (numéricas, *String* e booleanos) que representam os diferentes dados que podem ser guardados;
- Realizamos diferentes operações matemáticas em números e em nossas variáveis.

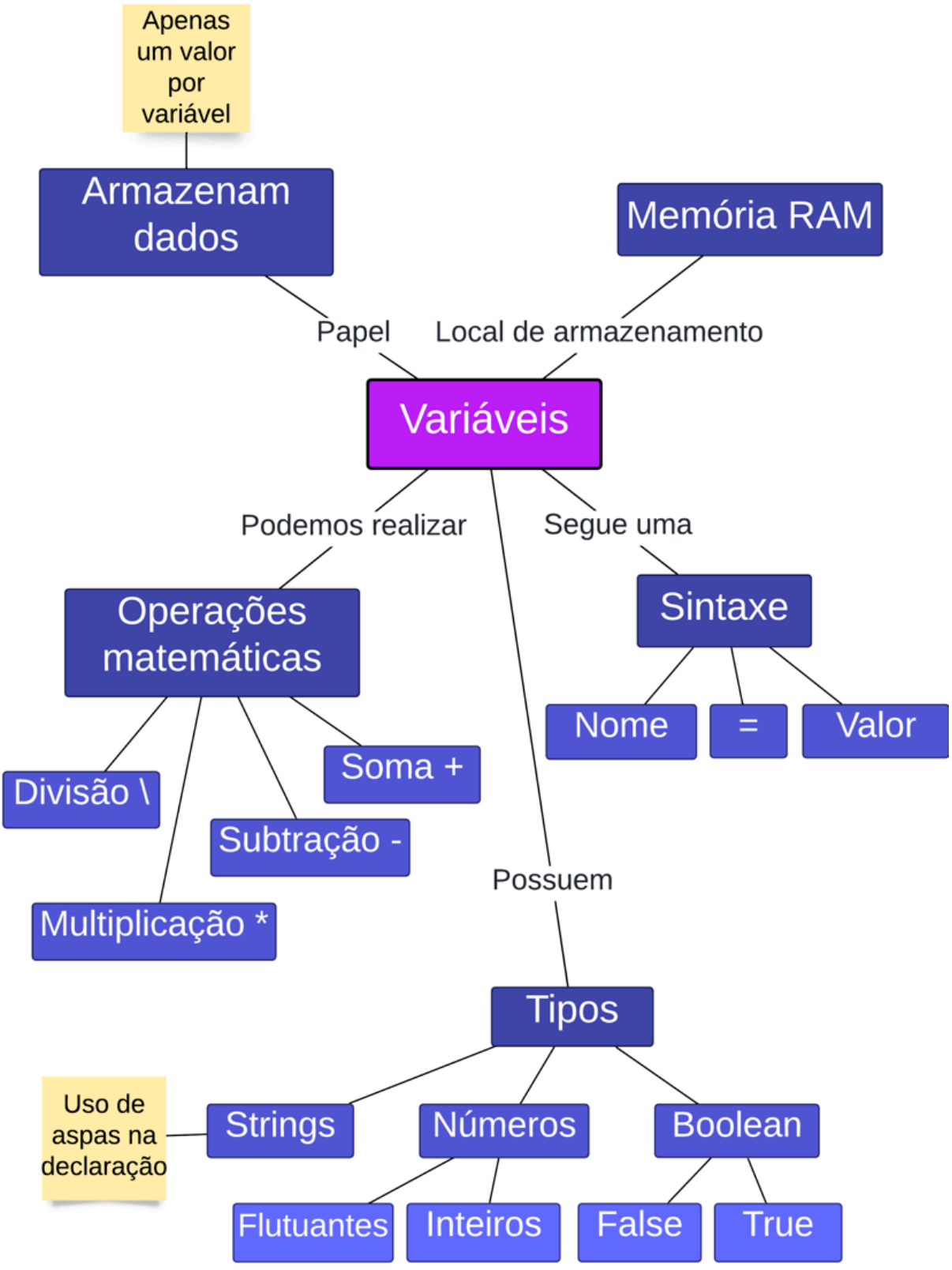
Apenas um valor por variável
Armazenam dados
Memória RAM
Papel
Local de armazenamento
Variáveis
Podemos realizar
Segue uma
Operações matemáticas
Sintaxe
Nome
=
Valor
Divisão \
Soma +
Subtração -
Multiplicação *
Possuem
Tipos
Uso de aspas na declaração
Strings
Números
Boolean
Flutuantes
Inteiros
False
True

Figura 4.15: Mapa mental do capítulo 4.

Capítulo 5
Erros... eles acontecem e são importantes

1. O que você já sabe?

A programação possui uma série de regras que define a forma como utilizamos seus recursos, denominada sintaxe. Por exemplo, para criar uma variável devemos informar o seu nome, seguido de um sinal de igualdade e do valor que será guardado. Tentar declarar uma variável seguindo outro formato ocasionará um erro.

2. O que veremos?

Os erros fazem parte do aprendizado de programação e vão ocorrer com frequência. Neste capítulo vamos aprender sobre eles, suas causas e como poderiam ser solucionados.

3. Por que é importante?

Os erros vão lhe acompanhar durante a programação, independente da sua experiência. O conhecimento apresentado neste capítulo ajudará os aprendizes no enfrentamento destes problemas. Para além disso, é fundamental entender o erro como parte do processo e uma oportunidade de melhoria.

O que você vai aprender

Ao término deste capítulo você compreenderá:

1. O que são erros na programação;
2. Os tipos de erros que podem ocorrer e suas causas;
3. Como o Python apresenta os erros em nossos algoritmos;
4. Identificar a solução para os principais erros.

5.1 Erros na programação

Os erros acontecem na programação com uma certa frequência. Os iniciantes vivenciam essa experiência de diferentes formas. Alguns se frustram e desistem, enquanto outros vão compreender que a existência de erros faz parte do processo de aprendizagem e utilizarão esse conhecimento para aperfeiçoar suas lacunas. Certamente o segundo grupo de estudantes apresentará maior chance de sucesso na programação.

O erro é um sinal de que algo precisa ser ajustado. Isso não quer dizer falta de conhecimento, falta de aptidão ou fracasso. O erro tão somente quer dizer "revise e corrija" seus últimos passos. Por isso criamos um capítulo exclusivamente para essa finalidade.

Muitos livros ignoram ou pouco discutem sobre os erros de programação. No entanto, em nossa experiência como professores, acreditamos que é fundamental saber reconhecer os erros ainda no começo do aprendizado. Sem dúvidas, isso ajuda o programador a encontrar a solução para sua correção.

A definição de um erro, de acordo com o dicionário de Oxford Languages, é:

> "ato ou efeito de errar;"

Assim, trazendo para o contexto de programação, precisamos compreender o que seria o ato de errar. Por exemplo, considere o algoritmo a seguir que apresenta um problema:

```
x == 10
```

A declaração da variável *x* está incorreta, pois utiliza dois sinais de igualdade e viola a sintaxe do Python. Situações como essa são conhecidas por **erro de sintaxe**.

Na maioria dos casos, o erro é resultado de um ato indevido realizado pelo programador. Em tese, ninguém comete um erro propositalmente. No exemplo, possivelmente quem programou não assimilou corretamente a sintaxe de declaração de variáveis ou simplesmente cometeu um erro de digitação.

Existem erros que ocorrem pela má interpretação de um problema que pode resultar em um algoritmo sem erro de sintaxe, mas que produz um resultado incorreto; ou por fatores externos que afetam nossos programas. Por exemplo, ao enviar uma mensagem de texto no WhatsApp desconectado da Internet. Neste caso, o programa não possui erros de sintaxe e foi escrito para produzir a resposta correta, mas a falta de Internet fez com que o programa não cumprisse seu objetivo. Esta última categoria é conhecida por erros em tempo de execução e não será abordada neste livro, por ser um tópico mais avançado.

5.2 Erros de sintaxe

Esse tipo de erro ocorre quando uma instrução de programação é escrita incorretamente e viola alguma das regras (sintaxe) do Python ou de outra linguagem de programação. Eles ocorrem com frequência com iniciantes que ainda não se apropriaram corretamente das regras existentes.

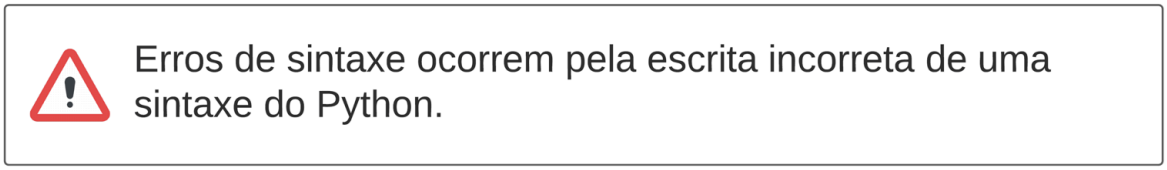

Figura 5.1: Erros de sintaxe ocorrem pela escrita incorreta de uma sintaxe do Python.

Interpretando as mensagens de erro

O código a seguir apresenta um outro tipo de erro de sintaxe.

```
nome = "Leonardo
```

Observe que apenas uma aspas foi utilizada para declarar a variável *nome* e sabemos que isso está incorreto, pois é necessário incluir uma aspas no começo do texto e outra ao final.

Ao executar o código acima o Python acusará a existência de um erro do tipo *SyntaxError* e apresentará a seguinte mensagem:

```
>>> nome = "Leonardo
  File "<stdin>", line 1
    nome = "Leonardo
                   ^
SyntaxError: EOL while scanning string literal
```

Figura 5.2: Mensagem de erro apresentada pelo Python.

Para iniciantes, a mensagem anterior pode assustar. Há muita informação nova e o texto está em inglês, mas ela possui indicações importantes que o ajudarão a resolver o problema em seu algoritmo.

A primeira delas é a indicação do local onde possivelmente encontra-se o erro (ressaltamos que por vezes essa indicação não é precisa). A mensagem também apresenta a linha onde está o problema (em inglês *line*) e o tipo de erro: *SyntaxError*, seguido de uma mensagem associada à causa do erro (falta de aspas).

É pouco provável que um iniciante vá compreender essas informações, o que nos motivou a escrever este capítulo.

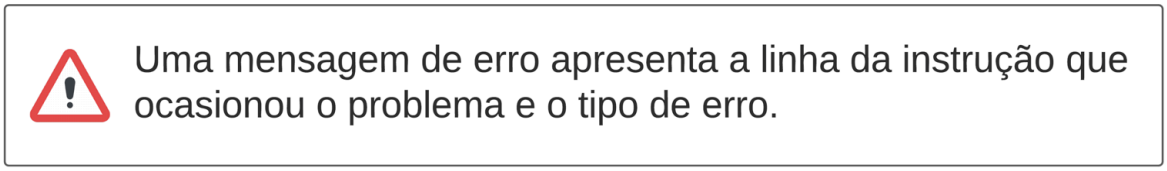

Figura 5.3: Uma mensagem de erro apresenta a linha da instrução que ocasionou o problema e o tipo de erro.

É comum que ao vivenciar uma situação de erro de sintaxe o estudante ignore a mensagem apresentada (por não conseguir interpretá-la) e tente resolver o problema por tentativa e erro (Safei, 2014). Essa estratégia não é eficaz e pode frustrar a pessoa, pois muitas vezes a solução para o problema não é óbvia.

Assim, é fundamental compreender o que a mensagem de erro informa, localizar as linhas que estão associadas ao problema, para a partir disso definir estratégias de solução. Apesar de aparentemente mais trabalhoso, estes passos ajudarão você a resolver qualquer problema, o que no longo prazo vai fazer de você um programador ou programadora mais eficaz.

Categorias de erros de sintaxe

Há uma variedade de situações que ocasionam erros de sintaxe. O Python subdivide esses erros em três categorias: *SyntaxError*, *NameError* e *TypeError*, a depender da causa do problema.

A primeira categoria representa problemas com a escrita de instruções que violam a sintaxe do Python, ilustrado no exemplo anterior. A segunda categoria refere-se a problemas com o uso de variáveis, como tentar utilizar uma variável não declarada. O terceiro ocorre quando se utilizam incorretamente os tipos das variáveis, por exemplo, ao somar um número com uma *String*.

As principais causas para erros de sintaxe serão apresentadas a seguir, juntamente a suas soluções. Não apresentaremos todos os

erros neste capítulo, pois alguns são relacionados a conteúdos que ainda não foram apresentados e sua discussão será realizada mais adiante.

SyntaxError

Erros nessa categoria indicam o uso incorreto da sintaxe do Python.

Problema 01	**Inverter a ordem na atribuição de dados a variáveis.**
Descrição	A ordem em que o nome e dado de uma variável são informados interfere na sua criação. Observe na instrução abaixo como o nome da variável está em uma ordem invertida.
Algoritmo incorreto	`2 = x`
Mensagem de erro	*SyntaxError: cannot assign to literal*
Solução	`x = 2`

Problema 02	**Declarar uma *String* sem aspas de abertura ou fechamento.**
Descrição	As Strings precisam ser declaradas com aspas de abertura e fechamento.
Algoritmo incorreto	`nome = "Leonardo`
Mensagem de erro	*SyntaxError: EOL while scanning string literal*
Solução	`nome = "Leonardo"`

Problema 03	**Ausência de números em operações matemáticas.**
Descrição	As operações matemáticas devem ser realizadas entre pares de números. No exemplo abaixo faltou um desses dados.
Algoritmo incorreto	`x = 2 +`
Mensagem de erro	*SyntaxError: invalid syntax*
Solução	`x = 2 + 3`

NameError

Erros nessa categoria envolvem o uso e declaração de variáveis. Alguns também são ocasionados por violação da sintaxe.

Problema 04	**Declarar uma variável com duas igualdades.**
Descrição	A sintaxe para declaração de uma variável requer o uso de apenas um sinal de igualdade. No exemplo abaixo tentou-se realizar esse procedimento com duas igualdades.
Algoritmo incorreto	`x == 2`
Mensagem de erro	*NameError: name 'x' is not defined*
Solução	`x = 2`

Problema 05	**Utilizar uma variável não declarada.**
Descrição	O nosso algoritmo é lido pelo computador de cima para baixo, assim, na execução da linha 1 ainda não há uma variável criada com o nome `y`, o que ocasionará um erro.
Algoritmo incorreto	`x = y`
Mensagem de erro	*NameError: name 'y' is not defined*
Solução	Antes de fazer uso de uma variável, ela precisa ser declarada.

TypeError

Erros nessa categoria ocorrem pela realização de operações entre variáveis que são de tipos diferentes.

Problema 06	**Realizar operações com variáveis de tipos diferentes.**
Descrição	Não é possível concatenar uma *String* com números ou somar números com *Strings*.
Algoritmo incorreto	`nome = "Leonardo, idade "+34`
Mensagem de erro	*TypeError: can only concatenate str (not "int") to str*
Solução	`nome = "Leonardo, idade 34"`

Há diferentes tipos de erro de sintaxe que são ocasionados pelas mais diversas causas. É fundamental compreender o que provocou o problema.

Figura 5.4: Há diferentes tipos de erro de sintaxe que são ocasionados pelas mais diversas causas. É fundamental compreender o que provocou o problema.

Localizando a causa do erro

Um dos grandes desafios para a pessoa programadora é localizar a instrução que ocasionou o erro. Isso requer um entendimento sobre o que pode desencadear o erro (como descrito anteriormente) e qual instrução está provocando isso. Esse processo se torna ainda mais difícil em um código com milhares de linhas.

Apesar de o Python oferecer uma indicação na mensagem de erro, nem sempre ela é precisa. Além disso, há situações onde a indicação está correta, mas a causa do problema está em linhas anteriores. Para entender isso é preciso relembrar que os algoritmos são executados de cima para baixo e o processamento das linhas anteriores afeta o resultado das linhas que ainda serão executadas. Vamos demonstrar este caso no exemplo a seguir, onde duas variáveis são somadas, mas uma delas é uma *String*:

```
x = 2
y = "2"
soma = x + y
```

A execução do algoritmo resultará na seguinte mensagem de erro:

```
Traceback (most recent call last):
  File "exemplo.py", line 3, in <module>
    soma = x + y
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Figura 5.5: Problema ao realizar uma operação entre variáveis de tipos diferentes.

Apesar de a indicação da linha do algoritmo que resultou neste problema estar correta, o problema está na linha anterior, onde a variável *y* foi declarada como *String* e não número.

Assim, ao ser avisado sobre um erro em seu código, observe a linha indicada pelo Python, mas também revise as linhas anteriores.

5.3 Erros lógicos

Existem situações onde o seu algoritmo não apresenta erros de sintaxe, mas não produz o resultado esperado. Veja o código a seguir que seria utilizado para converter de metros para centímetros:

```
metros = 1
centimetros = metros/100
```

Ao executar esse algoritmo o Python não indicará nenhum dos erros anteriores, pois de fato não há. Entretanto, o resultado da conversão será errado, pois deve-se utilizar a multiplicação e não divisão. Em casos como esse dizemos que o código está **logicamente incorreto**.

Os erros lógicos ocorrem por diferentes motivos. Pode acontecer de o(a) desenvolvedor(a) não saber como uma determinada operação deve ser realizada (como a conversão apresentada). Em outros casos pode ocorrer uma má interpretação do problema que está sendo resolvido. Por exemplo, uma escola pode solicitar um sistema para calcular os aprovados de um ano letivo e o programador considerar a nota mínima como sendo 6, quando na realidade é 7. Assim, torna-se fundamental entender corretamente o problema que será resolvido com a programação, o que nem sempre é um processo fácil.

O uso incorreto de recursos da programação também pode ocasionar erros lógicos. Não nos referimos ao uso indevido da

sintaxe, como apresentado anteriormente, mas uma falha na compreensão sobre como um recurso da programação deveria ser utilizado. A seguir exemplificamos este problema, onde o desenvolvedor representou o número 1.000 de uma forma incorreta. Na sequência ele realizou o cálculo de dez por cento deste valor.

```
valor = 1.000
percentual = 0.1
resposta = valor * percentual
```

Muitos aprendizes cometem esse erro, pois trazem essa representação numérica do dia a dia para a programação. No entanto, como discutido no capítulo anterior, sabemos que está incorreta.

Pelo fato de o Python não apontar a possível causa para os erros lógicos eles são mais difíceis de serem corrigidos. Utilizar fóruns de programação (como o Stack Overflow - https://pt.stackoverflow.com/) ou pedir ajuda a amigos é uma estratégia que pode ser utilizada caso não consiga localizar o problema.

Em resumo, a solução para os erros passa por: reconhecer o erro, refletir sobre seu conhecimento e se necessário, pedir ajuda (em fóruns, colegas, professores ou revisando o conteúdo de um livro como esse). Também é importante compreender que eles fazem parte do aprendizado, sendo natural cometê-los.

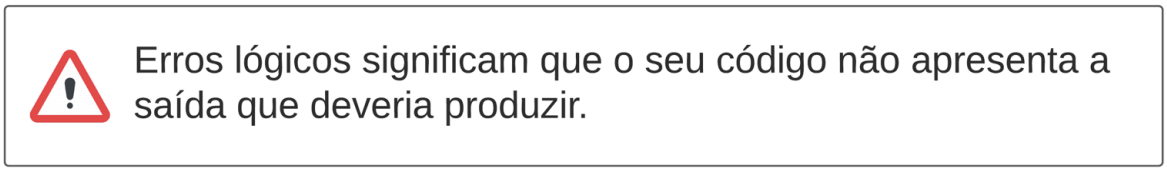

Figura 5.6: Erros lógicos significam que o seu código não apresenta a saída que deveria produzir.

Questões

1) Leia os códigos a seguir e indique qual tipo de erro de sintaxe (`SyntaxError` , `NameError` e `TypeError`) eles apresentam:

a.

```
x = 2
y = "4"
soma = x + y
```

b.

```
idade = "40
```

c.

```
livro = Programacao
```

2) O código a seguir foi construído para calcular uma média. Há algum problema? Se sim, qual seria a sua correção?

```
nota_um = 10
nota_dois = 5
nota_tres = 4
media = nota_um + nota_dois + nota_tres / 3
```

3) Analise a mensagem de erro a seguir e identifique as suas partes: trecho do código com problema, sua linha, o tipo de erro e a mensagem específica do erro.

```
Traceback (most recent call last):
  File "exemplo.py", line 3, in <module>
    soma = x + y
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Figura 5.7: Mensagem de erro apresentada pelo Python.

Respostas

1) Identifique os tipos de erros de sintaxe (`SyntaxError` , `NameError` e `TypeError`) dos códigos a seguir:

a.

```
x = 2
y = "4"
soma = x + y
```

R: TypeError

b.

```
idade = "40
```

R: SyntaxError

c.

```
livro = Programacao
```

R: NameError.

2) O código a seguir foi construído para calcular uma média. Há algum problema? Se sim, qual seria a sua correção?

```
nota_um = 10
nota_dois = 5
nota_tres = 4
media = nota_um + nota_dois + nota_tres / 3
```

R: O algoritmo apresenta um erro lógico, pois o cálculo da média requer que os valores sejam somados antes da divisão. Da forma como está no exemplo, a variável `nota_tres` está sendo dividida por 3 (três), e esse resultado, somado às duas outras variáveis.

3) Analise a mensagem de erro a seguir e identifique as suas partes: trecho do código com problema, sua linha, o tipo de erro e a mensagem específica do erro.

```
Traceback (most recent call last):
  File "exemplo.py", line 3, in <module>
    soma = x + y
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Figura 5.8: Mensagem de erro apresentada pelo Python.

R: O trecho com problema é: `soma = x + y`, que se encontra na linha 3. O resultado foi um erro do tipo `TypeError`, e a mensagem indica que não é possível realizar uma operação `+` entre inteiro e *String*.

5.4 Resumo e mapa mental

- Neste capítulo foram apresentados diferentes tipos de erros que ocorrem quando escrevemos nossos algoritmos;
- Existem três categorias de erros: sintaxe, lógica e erros em tempo de execução.
- Erros de sintaxe são ocasionados por instruções de programação escritas incorretamente. Para essa categoria de erro o Python apresenta uma mensagem indicando a causa do problema.
- Erros lógicos ocorrem quando o algoritmo está escrito sem erros de sintaxe, mas não produz o resultado esperado;
- Erros no tempo de execução ocorrem quando um algoritmo é lido pelo computador, mas alguma situação adversa ocorre, como a falta de internet, ou um dado inválido é inserido pelo usuário.

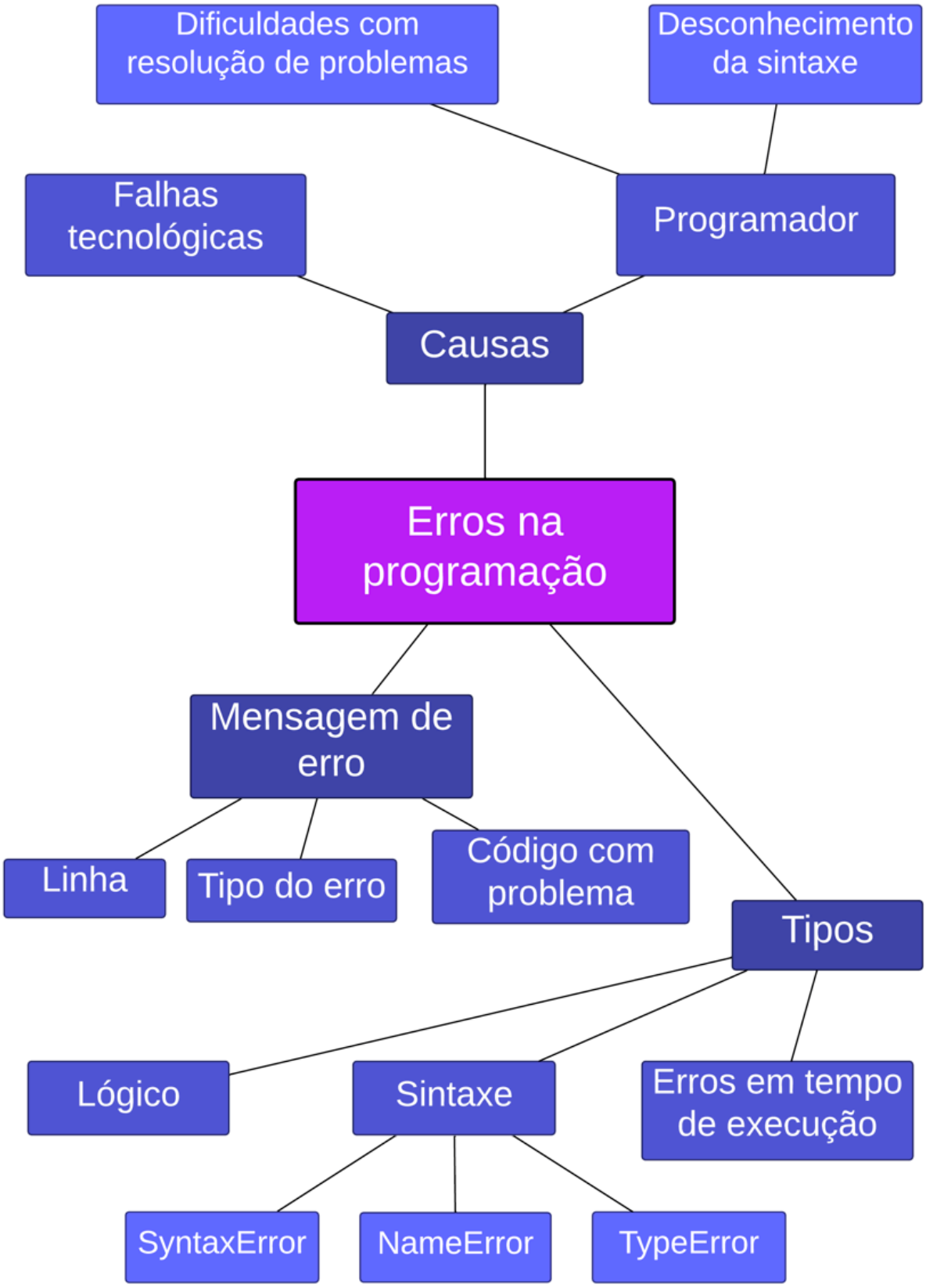
Dificuldades com resolução de problemas
Desconhecimento da sintaxe
Falhas tecnológicas
Programador
Causas
Erros na programação
Mensagem de erro
Linha
Tipo do erro
Código com problema
Tipos
Lógico
Sintaxe
Erros em tempo de execução
SyntaxError
NameError
TypeError

Figura 5.9: Mapa mental do capítulo 5.

Capítulo 6
Leitura e apresentação de dados

1. O que você já sabe?

Nossos programas manipulam constantemente variáveis que podem ser de diferentes tipos: números, *Strings*, booleanos, entre outros. Até o momento, nós declaramos as variáveis com um valor predefinido na sua criação. No entanto, no mundo real, é o usuário quem vai inserir alguns dados de nosso software.

2. O que veremos?

Neste capítulo vamos aprender como receber dados do usuário da nossa aplicação, um procedimento conhecido por **entrada de dados**. Por exemplo, quando você utiliza um aplicativo de calculadora e escolhe os números e a operação matemática que deseja realizar, esses dados representam entradas e são guardados em variáveis. Da mesma forma, precisamos de um meio para exibir informações ao nosso usuário (por exemplo, o resultado do cálculo), recurso que ainda não apresentamos e conhecido por **saída de dados**.

3. Por que é importante?

Entrada e saídas de dados são recursos essenciais, pois permitem que o usuário interaja com a aplicação e receba dela respostas que espera. Esse é um dos principais recursos que tornam os programas poderosos, pois eles podem dar respostas personalizadas a depender das entradas que lhe são passadas.

O que você vai aprender?

Ao término deste capítulo você compreenderá:

1. O que são entradas e saídas

2. Como realizar a leitura de dados com o Python;
3. Como utilizar as entradas para realizar cálculos matemáticos;
4. Como realizar a saída de dados com o Python;

6.1 Por que precisamos de entrada e saída de dados?

Agora que você já sabe o que são as variáveis é hora de avançar para outros conceitos da programação: entradas e saídas. Primeiramente é importante compreender quando utilizamos este recurso e vamos explicar a partir do código a seguir, escrito para realizar uma operação de soma:

```
numero_um = 5
numero_dois = 10
resultado_soma = numero_um + numero_dois
```

Toda vez que esse algoritmo for executado o valor da variável `resultado_soma` será 15. Isso acontece, pois os dados guardados nas variáveis `numero_um` e `numero_dois` não se alteram. No entanto, em uma aplicação de calculadora do mundo real, sabemos que esses valores são informados pelo usuário da aplicação, que pode alterá-los quando quiser.

Diante da limitação que os nossos códigos atuais apresentam, é preciso permitir que eles recebam dados inseridos pelo seu usuário. Antes de mostrar como realizar isso, é preciso esclarecer quais são as partes de uma aplicação. Esqueça por algum tempo a programação, pense como um usuário de um aplicativo, WhatsApp, por exemplo. Ao utilizá-lo você interage com a sua interface gráfica que é formada por botões, espaços para digitar textos, ícones, entre outros. Por trás de tudo isso está o que chamamos de código-fonte, que são as instruções de programação a que apenas a empresa WhatsApp possui acesso. Assim, podemos pensar em nossos

programas como uma estrutura em duas camadas: interface e código, como ilustrado na figura a seguir.

Figura 6.1: Exemplo de camadas de uma aplicação. Crédito da imagem: freepik

As entradas seguem o fluxo interface -> código, e isso faz sentido, pois os recursos de entrada estão na interface (botões, campos de texto, entre outros), as saídas fazem o caminho oposto. Algum processamento é feito no código e a sua resposta é apresentada na interface. Assim, percebe-se que o usuário interage com a interface gráfica do aplicativo, que repassa essas ações para a camada de código, responsável por processá-las. Por sua vez, a resposta então é apresentada na interface. Por exemplo, no WhatsApp, ao escrever um texto e pressionar no botão de envio de mensagem (ações que ocorrem na interface do aplicativo), espera-se que ela seja enviada ao contato desejado (a partir de um processamento realizado pelo código). O resultado disso é uma resposta produzida pela aplicação, por exemplo, exibindo um ícone para indicar que a mensagem foi encaminhada ao destinatário. Nesse exemplo temos os dois procedimentos descritos anteriormente: entrada e saída.

Apresentamos a organização de um programa como uma estrutura de duas camadas, o que é uma forma bastante simplista. Existem outras, como banco de dados e negócio, mas que pela sua complexidade fogem ao escopo deste livro.

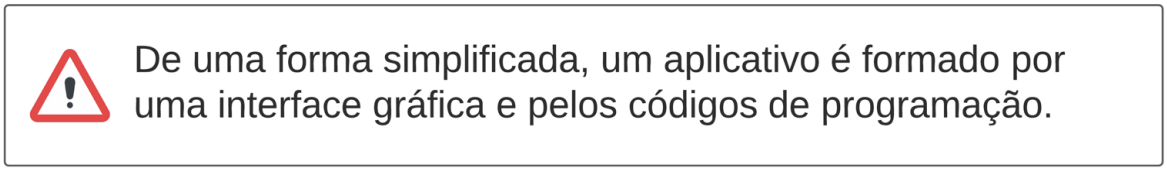

Figura 6.2: De uma forma simplificada, um aplicativo é formado por uma interface gráfica e pelos códigos de programação.

Com essa base teórica podemos avançar para compreender como utilizar entradas e saídas em nossos algoritmos.

6.2 Entrada de dados

A leitura de dados do teclado requer o uso de uma instrução especial da programação chamada de `input()`. Sua sintaxe é representada da seguinte forma:

```
variavel = input()
```

Veja um exemplo de uso a seguir:

```
nome = input()
```

Ao executar este código, o computador vai permitir que o usuário utilize o seu teclado. Tudo que for digitado até pressionar a tecla *ENTER* será armazenado na variável `nome`.

Entretanto, caso o usuário não digite nenhuma informação, o restante do algoritmo (se houver) não será executado. Quando uma linha em que a instrução `input` se encontra é processada, o computador aguarda o usuário digitar algo no teclado e pressionar *ENTER*. Vamos ilustrar essa situação com o algoritmo a seguir, que

recebe uma entrada do usuário e apresenta uma saída na tela com uma saudação com o seu nome.

```
nome = input()
saudacao = "Olá "+nome
```

Ao executar esse código, o computador vai aguardar a entrada do usuário. Enquanto ela não ocorrer, a segunda linha nunca será executada e consequentemente a operação de concatenação não será processada. Após ele digitar algo, por exemplo, "Davi" e pressionar *ENTER*, o computador continuará a execução do código e a variável *saudacao* armazenará o valor "Olá Davi".

É possível ter mais de uma instrução `input` para ler diferentes dados:

```
numero_um = input()
numero_dois = input()
```

Ao executar o código anterior, o usuário irá visualizar apenas uma tela preta com um pequeno cursor branco piscando. Para ele isso não quer dizer muita coisa, por isso é interessante informar sobre qual informação precisamos que ele/ela digite. Para isso podemos modificar o uso da instrução `input` para incluir um texto que será apresentado na tela do usuário no momento em que ele precisar escrever um dado. Para fazer isso vamos escrever uma *String* dentro dos parênteses do `input` :

```
numero_um = input("Digite o primeiro número")
numero_dois = input("Digite o segundo número")
```

A figura a seguir ilustra o que a execução desse algoritmo apresentará ao usuário.

```
>>> numero_um = input("Digite o primeiro número")
Digite o primeiro número
```

Figura 6.3: Execução do algoritmo Olá mundo.

Percebe-se como a entrada de dados é um importante recurso da programação. Ao analisar um problema, dedique um tempo para identificar quais são as entradas que o seu algoritmo vai receber.

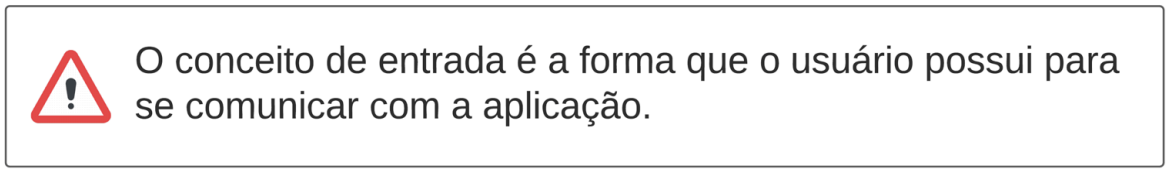

Figura 6.4: O conceito de entrada é a forma que o usuário possui para se comunicar com a aplicação.

Questões

1) Assinale Verdadeiro (V) ou Falso (F) para as afirmações:

```
(  ) A entrada de dados possibilita ao nosso algoritmo receber informações digitadas pelo usuário.
(  ) Nossos algoritmos são limitados a uma entrada de dados.
(  ) Para realizar a leitura de dados nós utilizamos a instrução `print`.
```

2) Crie um algoritmo para realizar a leitura de um dado e o armazene em uma variável.

3) Crie um algoritmo para realizar a leitura de um dado, informando ao usuário sobre o que ele precisa digitar.

4) Crie um algoritmo para realizar a leitura de dois textos do teclado e faça a sua concatenação.

Respostas

1) Assinale Verdadeiro ou Falso para as afirmações:

```
( V ) A entrada de dados possibilita ao nosso algoritmo receber informações digitadas pelo usuário.
( F ) Nossos algoritmos são limitados a uma entrada de dados.
( F ) Para realizar a leitura de dados nós utilizamos a instrução `print`.
```

2) Crie um algoritmo para realizar a leitura de um dado e o armazene uma variável.

```
informacao = input()
```

3) Crie um algoritmo para realizar a leitura de um dado, ao mesmo tempo que informa ao usuário qual informação está aguardando.

```
informacao = input("Escreva a informação que preciso")
```

4) Crie um algoritmo para realizar a leitura de dois textos do teclado e faça a sua concatenação.

```
nome = input()
sobre_nome = input()
nome_completo = nome + sobre_nome
```

6.3 Tipos e conversões de dados

Como vimos, existem diferentes tipos de dados que podem ser armazenados nas variáveis: números (inteiros e fracionários), Strings, booleanos, e outros que futuramente você conhecerá. Vamos agora compreender como isso se relaciona com o uso da instrução `input`. Observe o algoritmo a seguir:

```
numero_um = input()
```

O texto que o usuário digitar **sempre** será armazenado na variável como uma *String*. Portanto, se o usuário digitar o número 2, será guardado um texto "2" (sem as aspas). No entanto, isso ocasionará um problema em cálculos matemáticos. Veja o algoritmo a seguir:

```
numero_um = input()
numero_dois = input()
resultado_soma = numero_um + numero_dois
```

Dois dados são lidos do teclado e guardados nas variáveis `numero_um` e `numero_dois`. Vamos imaginar que foram digitados os números 2 e 3. Sabendo disso, consegue prever o valor da variável `resultado_soma`? **Se você pensou em 6 está enganado**. Vamos explicar, como a leitura de dados produz variáveis do tipo *String*, a operação realizada na terceira linha não é uma soma, mas sim concatenação. Portanto, o valor de `resultado_soma` será um texto "23".

Esse problema pode ser mais grave se tentarmos realizar uma operação matemática em uma *String*. Veja o exemplo a seguir:

```
numero_um = input()
subtracao = numero_um - 2
```

A execução do código produzirá o erro *TypeError: unsupported operand type(s) for -: 'str' and 'int'*. Não se assuste, o que o computador está tentando informar, de uma forma não muito amigável, é que você não pode realizar uma operação (neste caso de subtração) entre variáveis de tipos diferentes. E isso faz sentido, afinal, como diminuir 2 de uma *String*?

Assim, para realizar operações matemáticas em números lidos do usuário é preciso converter o dado de *String* para inteiro ou flutuante. Esse é um procedimento relativamente simples e que requer apenas a memorização de algumas instruções: **`int`** e **`float`**. Vamos conhecer a sintaxe de ambas:

```
variavel-inteira = int( variavel-a-ser-convertida )
```

```
variavel-flutuante = float( variavel-a-ser-convertida )
```

O funcionamento dessas instruções é simples, dentro dos parênteses informamos o dado que será convertido (uma *String* lida do teclado, por exemplo) e criamos uma variável para guardar o resultado desta

conversão, que será um inteiro se for utilizado o `int` ou um flutuante caso se utilize o `float`. Vamos ver na prática:

```
numero_string = input()
numero_inteiro = int( numero_string )
subtracao = numero_inteiro - 2
```

Observe que na linha 1 estamos lendo um dado do teclado e guardando na variável `numero_string`. Na sequência, cria-se uma variável nomeada `numero_inteiro`, que guardará o resultado da conversão de `numero_string` em inteiro, o que é feito com a instrução `int`. Após isso, podemos utilizar a variável `numero_inteiro` em operações matemáticas.

Também é possível converter de números para *String* com o uso da instrução **`str`**. Essa conversão em particular é importante, pois com frequência nós manipularemos números em nossos algoritmos, mas no momento de apresentá-los ao nosso usuário (saídas) essa informação será acompanhada de algum texto. Como não podemos concatenar números com textos, é preciso convertê-los para texto. Observe este caso no algoritmo a seguir:

```
numero_um = input()
numero_dois = input()
resultado_soma = numero_um + numero_dois
resultado_soma_texto = str( resultado_soma )
mensagem = "O resultado da soma é: "+resultado_soma_texto
```

Sem a conversão da variável `resultado_soma` para texto não seria possível concatená-la com um texto, pois o Python acusaria um erro do tipo *TypeError*.

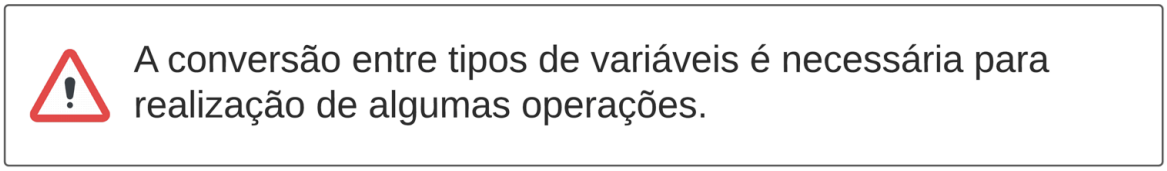

Figura 6.5: A conversão entre tipos de variáveis é necessária para realização de algumas operações.

6.4 Saída de dados

A saída da nossa aplicação, em geral, apresenta informações na tela do usuário. Sem esse recurso as aplicações não apresentariam os resultados de suas operações, fazendo com que essas percam parte do sentido de existirem. Já pensou uma calculadora que não apresenta o resultado final?!

Vamos compreender como apresentar uma saída a partir de um exemplo de soma matemática. No código a seguir, realizamos a leitura de dois dados do teclado, convertemos para inteiros e em seguida realizamos a operação de soma. Por fim, criamos uma mensagem personalizada para indicar o resultado do cálculo:

```
numero_um = input()
numero_dois = input()
resultado_soma = int(numero_um) + int(numero_dois)
resultado_soma_texto = str( resultado_soma )
mensagem = "O resultado da soma é: "+resultado_soma_texto
```

Você pode pensar que o usuário vai visualizar o valor da variável mensagem na tela, mas na realidade não vai. Lembra da estrutura em camada que a aplicação possui? a camada de código apenas é visível ao programador. Assim, para que uma informação seja apresentada na interface é preciso fazer uso de uma instrução chamada `print()`. Vamos conhecer a sua sintaxe:

```
print( dado-a-ser-apresentado )
```

O que é informado dentro dos parênteses é a informação que será apresentada na tela do usuário, podendo ser um dado ou o valor de uma variável. Vamos ver um exemplo:

```
print("Olá mundo!")
```

Ao executar esse algoritmo o computador apresentará na tela do usuário uma mensagem: `Olá mundo!` .

Também podemos utilizar a instrução `print` para apresentar os valores de nossas variáveis:

```
texto = "Olá mundo!"
print(texto)
```

Observe que ao utilizar o `print` com uma variável deve-se utilizar o seu nome sem aspas.

O `print` também pode exibir o valor de variáveis de diferentes tipos, como números e booleans:

```
print(2)
print(True)
```

É possível utilizar múltiplas instruções `print` em um único algoritmo:

```
texto = "Olá mundo!"
print(texto)

outra_texto = "Aproveite o livro!"
print(outra_texto)
```

A execução desse algoritmo resultará em dois textos impressos na tela, um abaixo do outro:

Figura 6.6: Saída do algoritmo anterior.

O `print` também apresenta uma outra sintaxe onde permite a exibição de múltiplos dados com uma única instrução:

```
print( dado_um, dado_dois, dado_tres )
```

Nesse caso, os dados serão impressos na tela, na mesma linha e separados por um espaço. Vamos visualizar um exemplo:

```
print("O número", 4, "é um número par e o número", 5, "é ímpar")
```

Ao executar este código o Python apresentará a saída: `O número 4 é um número par e o número 5 é ímpar .`

Vamos voltar ao exemplo da operação de soma e modificar o algoritmo para apresentar o resultado final ao usuário:

```
numero_um = input()
numero_dois = input()
resultado_soma = int(numero_um) + int(numero_dois)
resultado_soma_texto = str( resultado_soma )
mensagem = "O resultado da soma é: "+resultado_soma_texto
print(mensagem)
```

Há dois detalhes interessantes nesse algoritmo. O primeiro está na quarta linha, onde realizamos a conversão dos números para inteiro ao mesmo tempo que a operação de soma é realizada. Isso é possível na programação, pois o Python vai primeiro realizar as conversões. Na linha seguinte, convertemos o resultado do cálculo em uma *String*, procedimento necessário para a concatenação que é realizada na linha 5 (lembre-se, concatenação somente é permitida entre *Strings*). Após isso, o `print` vai se encarregar de apresentar o dado guardado em *mensagem* para o usuário.

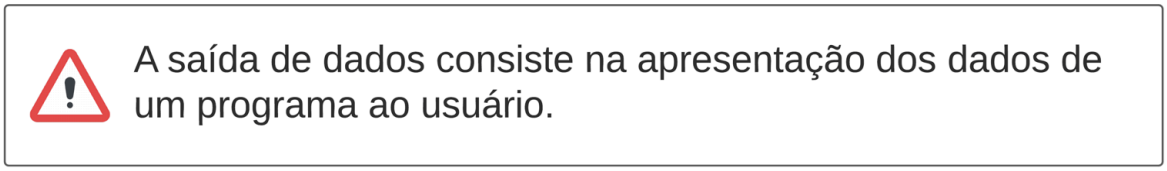

Figura 6.7: A saída de dados consiste na apresentação dos dados de um programa ao usuário.

6.5 Strings literais formatadas

Como visto no último exemplo, com frequência os textos apresentados ao usuário serão adaptados com o valor de alguma variável em específica. Naquela situação, nós concatenamos um texto base *O resultado da soma é:* com o resultado cálculo realizado.

Vamos ver outro exemplo, onde realizamos a leitura de um nome pelo teclado e a impressão de um texto de saudação:

```
nome = input()
print("Olá "+nome+", seja bem-vindo(a)")
```

Ao executar o programa, se o usuário escrever o nome Joaquim, o texto `Olá Joaquim, seja bem-vindo(a)` será apresentado. Isso foi possível por meio da concatenação de um texto com o valor armazenado na variável *nome*.

Apesar de o formato apresentado funcionar bem, desde a versão 3.6 do Python há um recurso que possibilita, dentre outras coisas, a combinação de textos e variáveis de uma forma mais elegante, denominado: *Strings literais formatadas*, também conhecido por *f-string*.

O mesmo exemplo, mas com o uso de uma *f-string* seria escrito da seguinte forma:

```
nome = input()
print(f"Olá {nome}, seja bem-vindo(a)")
```

Perceba que não precisamos concatenar, pois com f-strings podemos referenciar variáveis e instruções da programação diretamente em nossas *Strings*. Para isso, basta incluí-las dentro de { } (chaves), como no exemplo apresentado. Esse recurso é bastante útil, pois também possibilita a exibição de textos e números, também sem concatenação:

```
numero = 2
print(f"Este é o número: {numero}")
```

Podemos ainda formatar a apresentação dos números durante a sua exibição na tela, como limitar a quantidade de dígitos que serão apresentados:

```
numero = 2.3456789
print(f"Número com 2 casas decimais: {numero:.2f}")
print(f"Número com 3 casas decimais: {numero:.3f}")
```

Que resultará nas saídas:

```
Número com 2 casas decimais: 2.34
Número com 3 casas decimais: 2.345
```

Ao longo dos próximos capítulos falaremos sobre outros tipos de instruções que podem ser utilizadas em *f-strings*.

Questões

1) Assinale Verdadeiro ou Falso para as afirmações:

```
(  ) A saída de dados é uma forma de apresentar informações ao usuário.

(  ) A instrução `print` é usada para apresentar informações ao usuário.

(  ) É possível realizar operações matemáticas em variáveis do tipo
String.
```

( ) Para realizar a unificação de dados com Strings é preciso converter o dado em questão para String utilizando `str`.

2) Crie um algoritmo para realizar a leitura de um dado e o converta para inteiro.

3) O algoritmo de calculadora abaixo está incompleto, pois não apresenta o resultado da operação na tela do usuário. Escreva a instrução necessária para realizar esse procedimento:

```
numero_um = input()
numero_dois = input()
resultado_soma = int(numero_um) + int(numero_dois)
resultado_soma_texto = str( resultado_soma )
mensagem = "O resultado da soma é: "+resultado_soma_texto
```

Respostas

1) Assinale Verdadeiro ou Falso para as afirmações:

(V) A saída de dados é uma forma de apresentar informações ao usuário.

(V) A instrução `print` é usada para apresentar informações ao usuário.

(F) É possível realizar operações matemáticas em variáveis do tipo String.

(V) Para realizar a unificação de dados com Strings é preciso converter o dado em questão para _String_ utilizando `str`.

2) Crie um algoritmo para realizar a leitura de um dado e o converta para inteiro.

```
numero = input()
numero_inteiro = int( numero )
```

3) O algoritmo de calculadora abaixo está incompleto, pois não apresenta o resultado da operação na tela do usuário. Escreva a instrução necessária para realizar esse procedimento:

```
numero_um = input()
numero_dois = input()
resultado_soma = int(numero_um) + int(numero_dois)
resultado_soma_texto = str( resultado_soma )
mensagem = "O resultado da soma é: "+resultado_soma_texto
print(mensagem)
```

6.6 Erros mais comuns

Entrada e saída de dados é um dos primeiros assuntos aprendidos na programação e por isso alguns aprendizes cometem pequenos equívocos.

Confusão entre as instruções. É comum que iniciantes confundam a instrução `input` com `print`, acreditando que a primeira realiza a impressão de informações e a segunda a leitura de dados.

Esquecer de converter os tipos. Um outro erro frequente é tentar realizar operações em dados lidos com `input` sem antes convertê-los para tipos numéricos:

Código com problema:

```
numero = input()
soma = numero + 2
```

Código corrigido:

```
numero = input()
soma = int(numero) + 2
```

6.7 Resumo e mapa mental

- Neste capítulo nós aprendemos que os programas precisam de entradas e produzem saídas para o usuário final;
- A leitura de dados é feita utilizando a instrução `input`;
- Para produzir uma saída utilizamos a instrução `print`;
- Por fim, não é possível realizar concatenação ou operações matemáticas em variáveis de tipos diferentes.

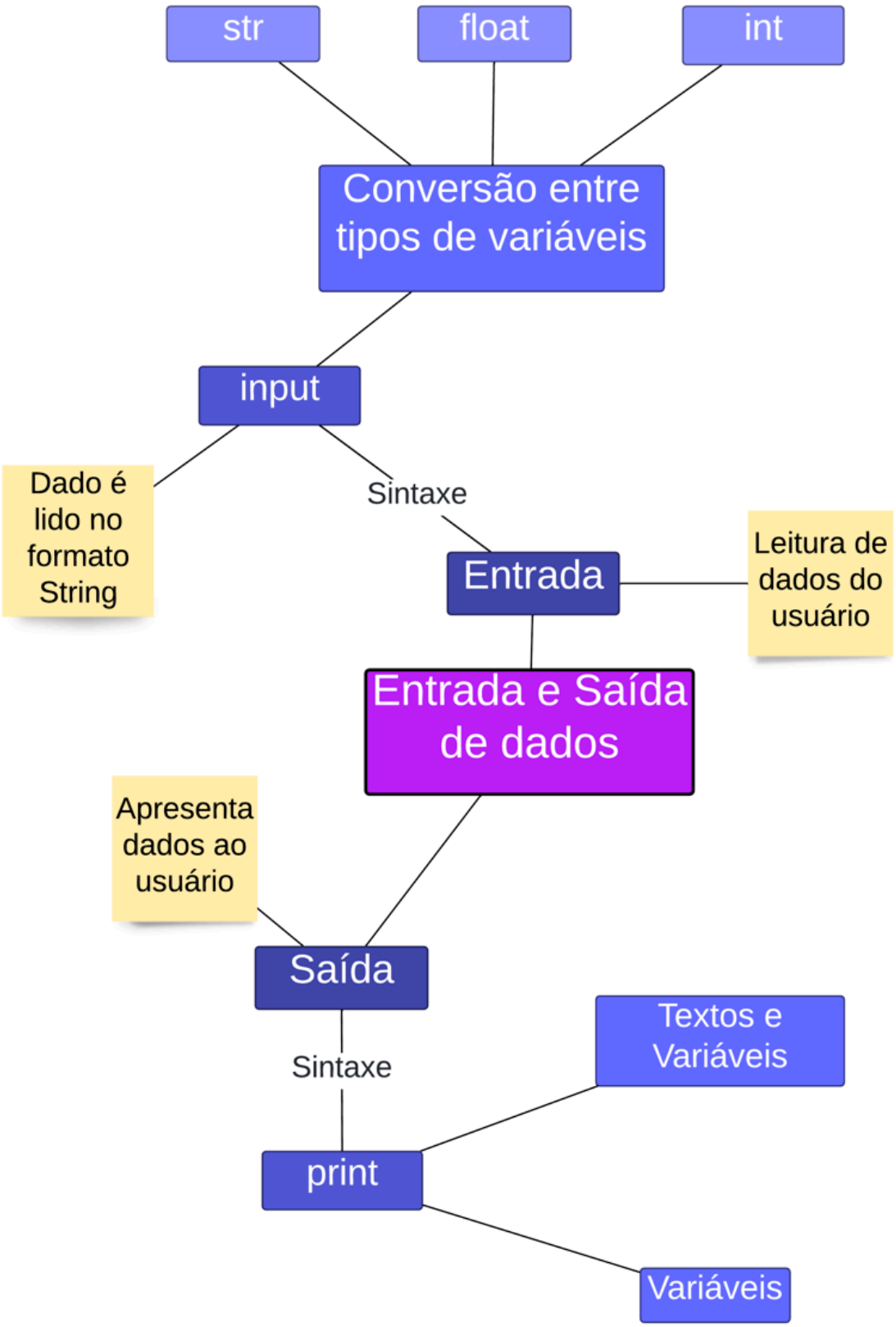
str
float
int
Conversão entre tipos de variáveis
input
Dado é lido no formato String
Sintaxe
Entrada
Leitura de dados do usuário
Entrada e Saída de dados
Apresenta dados ao usuário
Saída
Sintaxe
Textos e Variáveis
print
Variáveis

Figura 6.8: Mapa mental do capítulo 6.

Estruturas condicionais e de repetição

CAPÍTULO 7
Controlando o fluxo de execução do código com condições

1. O que você já sabe?

Nossos programas são formados por instruções que realizam algum tipo de processamento. Até o momento, aprendemos que o computador executa todas as instruções que estão no código de programação, mas nem sempre isso ocorre dessa forma.

2. O que veremos?

Há situações em que desejamos controlar quais instruções serão executadas pelos nossos programas, pois a realização de determinadas ações depende de alguma condição. Por exemplo, um saque em um caixa eletrônico bancário somente deve ser atendido se houver saldo na conta. Controles como esse são realizados por meio de um conceito da programação conhecido por condições.

3. Por que é importante?

As condições estão presentes em nosso dia a dia. Seja no banco, em um supermercado ou no seu trabalho. Essas restrições também estão presentes em todos os softwares, o que faz esse recurso ser muito importante para a programação.

7.1 As condições em nosso dia a dia

Para compreender a importância das condições na programação precisamos entender como elas também estão presentes em nossas vidas. Diariamente precisamos tomar diferentes decisões: ir ou não assistir a um filme no cinema, acelerar ou frear um carro, fazer ou não uma compra. A realização dessas ações depende de algumas **condições**.

Por exemplo, um cliente de um banco somente pode sacar o dinheiro no limite do seu saldo bancário. Se essa condição (possuir o dinheiro na conta) não for satisfeita, todas as ações vinculadas ao saque (a máquina liberar o dinheiro, diminuir do saldo esse valor, entre outras) não serão realizadas.

O software que está presente no caixa bancário controla a liberação do dinheiro por meio das condições. Como gostamos de explorar os significados das palavras, vamos ver o que o dicionário Cambridge fala sobre uma condição:

> "Expressa a ideia de que uma coisa **DEPENDE** de outra coisa"

Perceba o destaque na palavra depende, pois é exatamente assim que as condições funcionam. Com elas, a pessoa programadora pode criar uma verificação e definir as instruções de programação que serão processadas a partir disso. Assim, dizemos que a execução de algumas ações depende de alguma condição ser satisfeita.

7.2 Sintaxe para criação das condições

O primeiro passo para utilizar as condições é saber reconhecer quando elas serão necessárias. Na prática, isso será determinado

pelo problema que você vai desenvolver, pois as condições de um jogo de tiro são diferentes das presentes em um software de gestão de pontos de venda em um supermercado. Quando reconhecemos situações que dependem de algum tipo de decisão, há uma grande chance de sua implementação demandar uma condição.

Um outro elemento importante para entender as condições é o conceito de **estado de um programa**. Pense nele como uma 'fotografia' que representa como os dados de um programa se encontram em um determinado momento. Por exemplo, quando o cliente utiliza um caixa eletrônico e solicita um saque, as variáveis deste software possuem um conjunto de valores, como o saldo bancário naquele exato momento, o valor de saque requisitado, entre outros. Se ele utilizar esse caixa em um outro momento (um mês depois, por exemplo) pode ser que os dados sejam outros, pois possivelmente seu saldo e valor de saque serão outros.

Podemos então dizer que utilizamos as condições para instruir o computador a decidir quais instruções deve realizar. Isso é feito a partir de um critério criado pelo(a) programador(a) e que utiliza os dados armazenados nas variáveis.

Antes de apresentar a sintaxe de uma condição no Python, vamos mostrá-la utilizando o nosso português, pois ajudará na sua compreensão dos conceitos base. A seguir apresentamos a estrutura de uma condicional exemplificada a partir da operação de saque bancário:

> SE o valor de saque é MENOR OU IGUAL ao valor do saldo bancário da conta, ENTÃO realizar operação de saque.

Há alguns elementos nessa frase que merecem nossa atenção. A primeira é a palavra *SE*, que é utilizada para expressar a ideia de uma condição. Em seguida, estamos fazendo uma comparação entre as variáveis *valor de saque* e *valor do saldo bancário*. Neste caso,

nos interessa saber se o saque é menor ou igual ao saldo na conta, pois é o critério estabelecido pelo banco para liberar o saque.

Essa comparação pode resultar em verdadeiro ou falso, a depender do estado do programa. Quando verdadeiro, o computador vai executar o conjunto de instruções dependentes da condição. Por exemplo, os procedimentos para liberar o dinheiro em um caixa eletrônico, como diminuir o valor do saque do saldo bancário, liberar as notas em papel, apresentar uma mensagem indicando que o saque foi realizado com sucesso.

Esta é a grande característica de uma condição, pois permite ao(a) programador(a) especificar um critério que será utilizado para determinar quando um conjunto de instruções será executado.

7.3 Representando condições na programação

Agora que sabemos o que são condições, quando as utilizar e as suas partes, vamos ver como representá-las no Python. Sua sintaxe é escrita da seguinte forma:

```
if comparação:
    # instruções que serão executadas quando a condição for verdadeira
```

Há algumas regras do Python que precisamos seguir. Primeiro, a palavra `if` precisa estar em minúsculo. Além disso, você não pode ter nenhuma variável com esse nome, se não o Python acusará que há um problema. Existem outras restrições para nomes de variáveis e falaremos à medida que elas forem aparecendo.

A comparação é realizada entre um par de variáveis onde se avalia se uma é menor, maior, igual ou diferente da outra (há ainda operações podem ser realizadas e serão apresentadas em breve). Esse ponto é fundamental, pois é o resultado desta comparação que

será utilizado pelo computador para definir se deve ou não executar um conjunto específico de instruções.

Após a comparação há um sinal de `:` (dois pontos) e, nas linhas abaixo, deve-se escrever as instruções que serão executadas quando a condição for verdadeira.

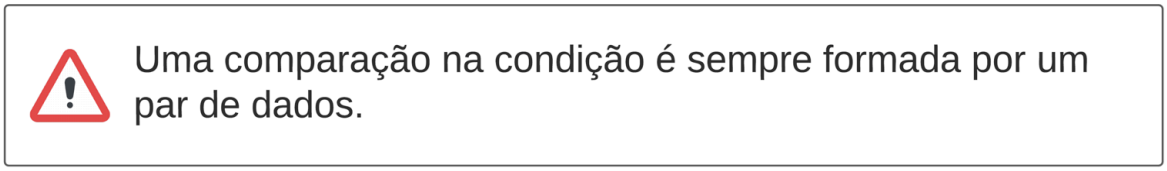

Figura 7.1: Uma comparação na condição é sempre formada por um par de dados.

No próximo exemplo vamos implementar a operação de saque utilizando o Python. Há duas variáveis, uma para guardar o valor que o cliente solicitou para saque (`valor_saque`) e outra que armazena o saldo bancário (`valor_saldo`):

```
if valor_saque <= valor_saldo:
    valor_saldo -= valor_saque
    print("Saque realizado com sucesso")
```

Observe que na comparação utilizamos os operadores matemáticos que representam maior que, menor que, igualdade, entre outros. Nesse exemplo em especial estamos combinando o menor que com o igual que, pois assim garante-se que saques menores que o valor do saldo ou que sejam iguais, serão atendidos.

Após a comparação estão as instruções que são executadas quando o seu resultado é verdadeiro. A primeira instrução atualiza o saldo do cliente, reduzindo dele o valor solicitado de saque. Na sequência, utiliza-se o `print` para indicar que a operação foi realizada com sucesso. Observe que a escrita dessas instruções está com um espaçamento no início, em comparação à linha do `if` . No Python, é necessário escrever dessa forma (que se chama **indentação**), pois

é assim que a linguagem reconhece quais instruções pertencem à condição, ou seja, aquelas que somente serão executadas quando a comparação resultar em verdadeiro. Assim, todas essas linhas de código devem estar indentadas. Caso essa regra da indentação seja descumprida, o Python acusará um erro do tipo **IndentationError**.

O conjunto de instruções que serão executadas pela condição é chamado de **escopo da condição**. O entendimento das condições depende da compreensão sobre como o Python executa essas instruções. O que estiver fora do escopo da condição será executado independentemente do resultado da comparação. No exemplo a seguir ilustramos isso, com instruções que fazem parte do escopo e outras fora dele.

```
print("Início do código")
saldo = 150
saque = int( input() )

if saque <= saldo:
    saldo = saldo - saque
    print("Saque realizado com sucesso")

print("Obrigado por utilizar nossos serviços")
```

As primeiras linhas do código estão fora do escopo da condição e por isso serão executadas independente do resultado da comparação. Observe na terceira linha como combinamos o uso das instruções `int` e `input`, isso é possível na programação. O Python vai garantir primeiro a execução de `input`, e o resultado disso será passado para que `int` realize a sua conversão para inteiro.

Esse procedimento é necessário, pois na comparação da condição estamos interessados em saber se um número é menor ou igual ao outro. Na sequência, estão as instruções que fazem parte do escopo da condição e somente serão executadas a depender do valor de saque informado pelo usuário.

A seguir apresentamos alguns cenários para esse exemplo e o que seria apresentado na tela em cada um deles:

Valores das variáveis	Resultado da verificação	O que aparecerá na tela:
saldo = 150 saque = 50	Verdadeiro	Saque realizado com sucesso Obrigado por utilizar nossos serviços
saldo = 150 saque = 200	Falso	Obrigado por utilizar nossos serviços.

Por fim, observe que a última instrução está sem indentação e consequentemente fora do escopo da condição. Assim, ela será executada independentemente da comparação que foi realizada.

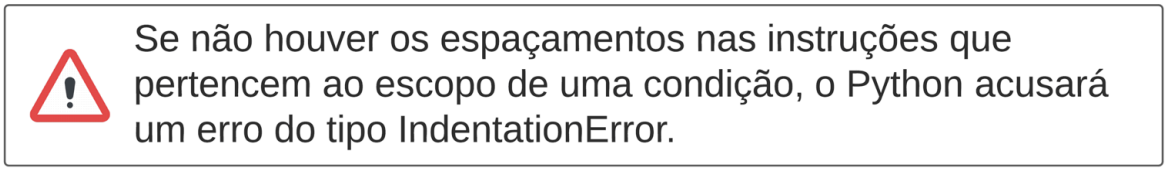

Figura 7.2: Se não houver os espaçamentos nas instruções que pertencem ao escopo de uma condição, o Python acusará um erro do tipo IndentationError.

Questões

1) Qual a sintaxe para representar uma condição em Python?

2) Em quais situações devemos utilizar condicionantes?

3) O que é o escopo de uma condição?

Respostas

1) Qual a sintaxe para representar uma condição na programação em Python?

```
if comparação:
    # escopo
```

2) Em quais situações devemos utilizar condicionantes na programação?

R: Para assegurar que um conjunto de instruções somente serão executadas quando determinados critérios forem atendidos.

3) O que é o escopo de uma condição?

R: O escopo define quais instruções pertencem a uma condição. Seriam as instruções a serem executadas quando a verificação da condição resultar em verdadeiro.

7.4 Comentários na programação

Em um dos exemplos deste capítulo utilizamos uma instrução que ainda não havia sido apresentada: `#`

Esse recurso é conhecido por comentário e permite que o(a) programador(a) escreva textos no código. Isso é útil quando precisamos fazer anotações que podem auxiliar na compreensão do algoritmo. Ilustramos esse recurso no código a seguir que apresenta o cálculo da energia cinética.

```
massa = float( input() )
velocidade = float( input() )
# fórmula de energia cinética: EC = m x v² / 2
energia = (massa * velocidade ** 2) / 2
```

Observe a parte do código que contém um `#` . Esse recurso foi usado para explicar a fórmula física utilizada para cálculo da energia cinética. Na prática um comentário não realiza nenhuma ação em nosso código e o interpretador Python o ignora.

Um outro trecho interessante desse código está no cálculo da energia onde utilizamos o operador `**`. Ele é utilizado para definir um número expoente, neste caso o número 2 passa a ser o expoente da variável `velocidade`.

Para comentários com mais de uma linha pode-se utilizar múltiplos `#`, sendo esta a forma recomendada pelo Python PEP 8, um conjunto de recomendações para escritas de códigos nesta linguagem:

```
# um comentário
# outro comentário
# mais outro comentário
```

Também é possível utilizar a sintaxe `"""`, conhecida por *String literals*:

```
"""
um comentário
outro comentário
"""
```

7.5 Operadores relacionais

Voltando a falar de condições... Aprendemos que elas realizam uma comparação entre um par de dados por meio de um operador matemático. Muitos já são conhecidos por você, pois aprendemos na escola.

A única diferença no uso desses operadores na programação é a forma como os sinais são representados. Veja na tabela a seguir:

Comparação	Representação matemática	Representação na programação	Exemplo
Maior que	>	>	10 > 5
Menor que	<	<	5 < 3
Maior ou igual que	≥	>=	5 >= 3
Menor ou igual que	≤	<=	3 <= 5
Igual	=	==	3 == 3
Diferente	≠	!=	3 != 5

Atenção, pois se errar na escolha do operador poderá ter um problema em seu código, como ilustrado no código a seguir. Nele, representamos a operação de saque, mas ao invés de utilizar o símbolo `<=` (menor ou igual que), utilizamos o `<` (menor que).

```
saldo = 50
saque = 50

if saque < saldo:
    saldo = saldo - saque
    print("Saque realizado com sucesso")
```

Apesar do valor solicitado para saque possibilitar a operação, esse código não vai executar o escopo do `if`. Esse problema vai ocasionar um **erro lógico**. Falamos sobre eles no capítulo 5.

Sempre que utilizar os operadores maior/menor e igual, o símbolo de > ou < deve vir antes da igualdade: >= ou <=. Se você escrever => ou =<, o Python não vai compreender.

Figura 7.3: Sempre que utilizar os operadores maior/menor e igual, o símbolo de > ou < deve vir antes da igualdade: >= ou <=. Se você escrever: => ou =<, o Python não vai compreender.

Com exceção dos operadores de igualdade e diferença, os demais somente podem ser utilizados com variáveis numéricas. É importante compreender isso, pois se você realizar a seguinte operação:

```
x = 4
b = "3"
if x > b:
    print("Maior")
```

Resultará em um erro: `TypeError: '>' not supported between instances of 'int' and 'str'`, pois a variável `b` não é numérica e sim uma *String*.

A comparação de uma condição utiliza duas igualdades (==). Não confundir com o operador de atribuição (=) que é usado para definir o valor de uma variável.

Figura 7.4: A comparação de uma condição utiliza duas igualdades (==). Não confundir com o operador de atribuição (=) que é usado para definir o valor de uma variável.

A seguir apresentamos o uso do operador de igualdade em uma comparação com *String* e booleano:

```
nome = input()

if nome == "Ada Lovelace":
    print("A primeira programadora da história!")
```

A comparação realizada neste código é bastante comum e utilizada para saber se o valor de uma *String* é igual a um texto em específico. No código, se o usuário digitar Ada Lovelace, aparecerá uma mensagem na tela *A primeira programadora da história!*. Atenção, pois o Python diferencia letras maiúsculas das minúsculas. Portanto, se o usuário digitar *ada lovelace* nada acontecerá.

No exemplo a seguir realizamos uma comparação entre uma variável booleana com o valor booleano `True` .

```
cinema_aberto = True

if cinema_aberto == True:
    print("Cinema está aberto")
```

Há outra sintaxe mais simplista para realizar essa comparação e que funciona somente com variáveis do tipo booleanas. Veja no exemplo a seguir:

```
cinema_aberto = True

if cinema_aberto:
    print("Cinema está aberto")
```

Observe que no *if* não utilizamos um par de dados na comparação, como no exemplo anterior. O Python considera essa sintaxe como o equivalente ao anterior.

Uma outra característica das variáveis booleanas é que o valor True equivale ao 1 e False ao 0. Isso é verificável pelo código a seguir:

```
dado = 1
if dado == True:
    # escopo quando a variável dado é verdadeira

outro_dado = 0

if outro_dado == False:
    # escopo quando a variável outro_dado é falso
```

Questões

1) Escreva o operador relacional adequado para responder às questões a seguir:

```
(     ) Igualdade
(     ) Maior que
(     ) Menor que
(     ) Diferente
(     ) Maior
(     ) Menor
```

2) Qual o resultado (Verdadeiro ou Falso) das comparações a seguir:

Comparação	Resultado
3 <= 4	
4 > 4	
5 > 6	
89 < 88	
"a" != "A"	
"a" == "a"	

Respostas

1) Escreva o operador relacional adequado para responder às questões a seguir:

```
(  ==  ) Igualdade
(  >=  ) Maior que
(  <=  ) Menor que
(  !=  ) Diferente
(   >  ) Maior
(   <  ) Menor
```

2) Qual o resultado (Verdadeiro ou Falso) das comparações a seguir:

Comparação	Resultado
3 <= 4	Verdadeiro
4 > 4	Falso
5 > 6	Falso
89 < 88	Falso
"a" != "A"	Verdadeiro
"a" == "a"	Falso

Comentário: observe que a comparação "a" != "A" resultará em True, pois estamos comparando um ‘a’ minúsculo com o ‘A’ maiúsculo.

7.6 Uso do operador else

Aprendemos que a comparação em uma condição resultará em verdadeiro ou falso. Até o momento trabalhamos apenas com condições que possibilitam uma resposta quando a comparação for verdadeira. Entretanto, as diversas situações do dia a dia podem demandar respostas quando a comparação for falsa. Vamos analisar essa situação por meio do exemplo do saque no caixa bancário:

```
saldo = int( input() )
saque = int( input() )

if saldo >= saque:
    saldo = saldo - saque
    print("Saque realizado com sucesso")
```

O código contempla apenas a situação em que o cliente pode realizar o saque. No entanto, caso isso não seja possível é preciso informá-lo sobre o problema. Assim, precisamos definir o conjunto de instruções que serão executadas quando a comparação não for verdadeira. Para isso, será utilizado a instrução `else`, que traduzindo para o português seria **SE NÃO**. Veja a sua sintaxe:

```
if comparação:
    # escopo do if
else:
    # escopo do else
```

Incluímos o `if` propositalmente, pois o `else` somente existe acompanhado dele. O `if` por sua vez pode existir sem um `else`. Podemos ler essa instrução da seguinte forma:

> SE << comparação >> resultar em verdade ENTÃO as ações do IF serão executadas; SENÃO outras ações serão executadas.

Vamos ver o exemplo anterior modificado para utilizar essa instrução:

```
saldo = int( input("Digite o saldo bancário") )
saque = int( input("Digite o valor de saque") )

if saldo >= saque:
    saldo = saldo - saque
    print("Você realizou um saque com sucesso.")
else:
    print("Você não possui saldo suficiente para realizar essa operação.")
```

É preciso entender que as instruções `if` e `else` são excludentes, ou seja, apenas o escopo de uma delas será executada. Isso faz sentido, pois a comparação realizada no `if` **sempre resultará em dois estados**: verdadeiro (disparando a execução do escopo do `if`) ou falso (fazendo com que o escopo do `else` seja executado).

Observe que a instrução else não exige uma verificação de condição, pois seu escopo sempre é executado quando a comparação do if resultar em falso.

Figura 7.5: Observe que a instrução else não exige uma verificação de condição, pois seu escopo sempre é executado quando a comparação do if resultar em falso.

Para ajudar na compreensão do `if` e `else` utilizaremos um recurso gráfico conhecido por Fluxograma. Veja a seguir:

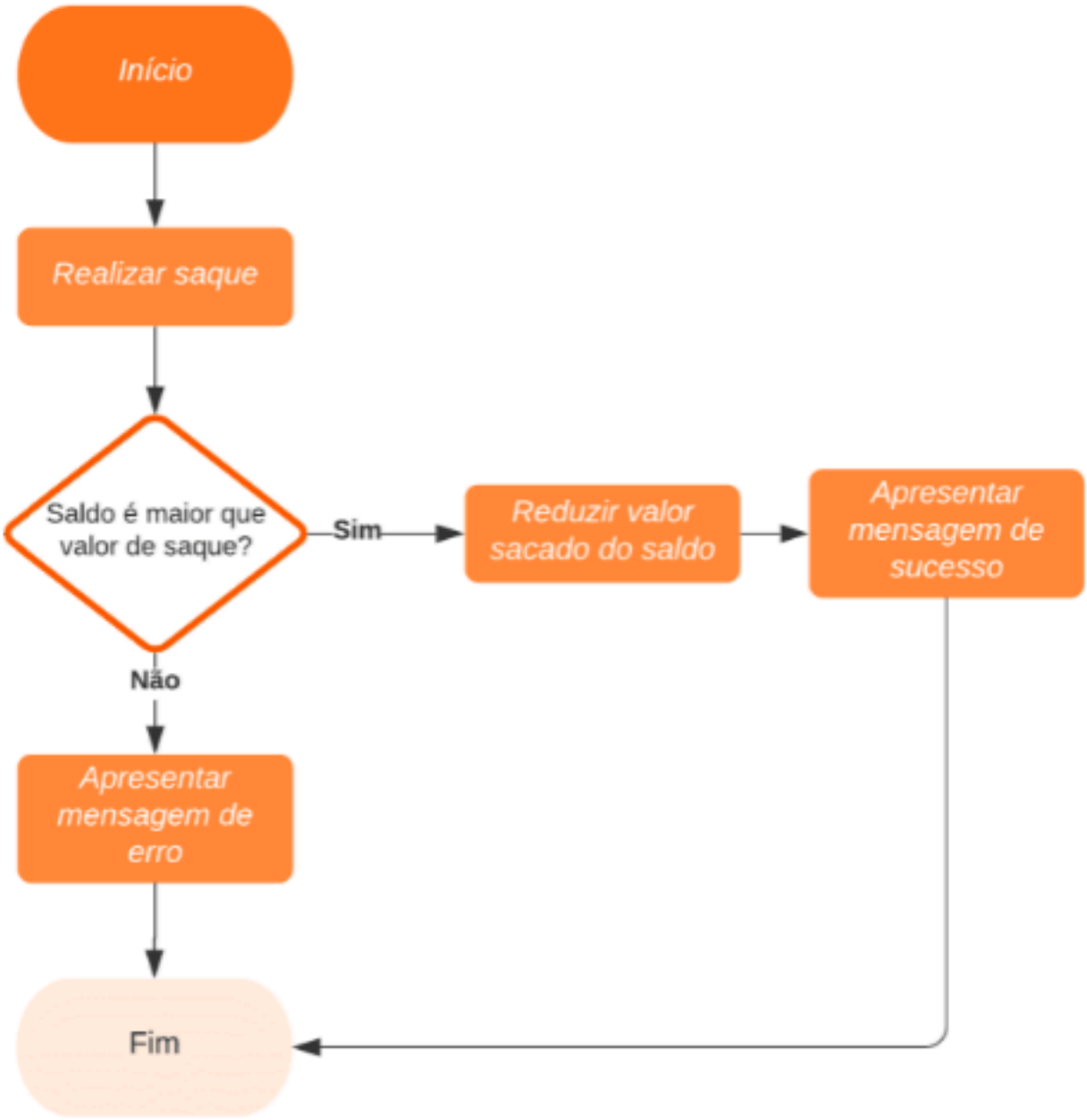

Figura 7.6: Fluxograma de execução do algoritmo anterior.

O gráfico demonstra a sequência de ações implementadas no código anterior. Inicia-se pela intenção de saque, que segue para uma análise que vai definir se essa ação será ou não executada (losango). Quando o resultado é verdadeiro executa-se um conjunto específico de instruções, sendo falso, outras instruções serão executadas.

Questões

1) Qual o nome da instrução utilizada para indicar que uma ação deve ser executada quando a verificação de uma condição resultar em falso?

2) Considere o código a seguir e indique o que será apresentado na tela:

```
x = 4

if x > 4:
    print("X é maior que 4")
else:
    print("X não é maior que 4")
```

3) Escreva um algoritmo que calcula a média de um estudante a partir de duas notas. Deve-se exibir na tela se ele foi aprovado (nota maior ou igual a 6) ou reprovado.

Respostas

1) Qual o nome da instrução utilizada para indicar que uma ação deve ser executada quando a verificação de uma condição resultar em falso?

R: A instrução chama-se `else`.

2) Considere o código a seguir e indique o que será apresentado na tela:

```
x = 4

if x > 4:
    print("X é maior que 4")
else:
    print("X não é maior que 4")
```

R: Será apresentado na tela: X não é maior que 4, pois a comparação `x > 4` resultará em falso, uma vez que o valor de x é 4

e esse valor não é maior que 4.

3) Escreva um algoritmo que calcula a média de um estudante a partir de duas notas. Deve-se exibir na tela se ele foi aprovado (nota maior ou igual a 6) ou reprovado.

```
nota1 = int( input("Digite a primeira nota") )
nota2 = int( input("Digite a segunda nota") )
media = (nota1+nota2)/2
if media >= 6:
    print("Aprovado")
else:
    print("Reprovado")
```

7.7 Múltiplas comparações em uma condição

As condições apresentadas nos exemplos envolveram apenas uma comparação por vez. No entanto, há casos onde a decisão sobre executar um conjunto de instruções depende do resultado de múltiplas comparações. Por exemplo, para sacar dinheiro não basta ter o saldo suficiente. Alguns bancos também verificam se o cliente não excedeu o limite de saques por dia, que em geral é de R$ 1.000. Assim, as duas condições precisam ser atendidas para que a operação de saque seja realizada. Poderíamos representar esse exemplo da seguinte forma:

> SE saldo for maior ou igual ao valor de saque E valor sacado no dia for menor ou igual a R$ 1.000 ENTÃO liberar saque.

Observe que entre as duas comparações temos a letra `E`, que tem o papel de conjunção, pois indica que as duas comparações precisam ser verdadeiras para que as instruções de saque sejam executadas.

Na programação temos um operador que realiza um papel semelhante e pertence ao grupo de **operadores lógicos**. Esses operadores asseguram que a execução do escopo da condição somente ocorrerá quando duas comparações forem analisadas. Inicialmente você precisará conhecer apenas dois desses operadores:

Operador	Descrição de uso
and	Utilizado quando se deseja que o escopo somente seja executado quando todas as comparações foram verdadeiras.
or	O escopo da condição é executado quando ao menos uma das comparações for verdadeira.

7.8 Operador and

O uso do operador `and`, que significa `e` em português, garante que as ações de uma condição somente serão executadas se as múltiplas comparações forem verdadeiras. Se apenas uma delas for falsa, o escopo da condição será ignorado.

Dando continuidade ao exemplo do saque no caixa bancário, vamos incluir outra comparação à nossa condição para representar a restrição no limite de retiradas diárias. Para isso, criaremos uma variável `valor_sacado` que guarda o total de dinheiro retirado em um dia. Para fins de exemplificação, ela será inicializada em 900. Observe o seu uso em conjunto com o operador `and`:

```
saldo = int( input("Digite o saldo bancário") )
saque = int( input("Digite o valor de saque") )
valor_sacado = 900 + saque
```

```
if saldo >= saque and valor_sacado <= 1000:
    # instruções para liberar o saque
```

O `and` também pode ser utilizado múltiplas vezes em uma única condição. A seguir, incluímos mais uma comparação para garantir que o valor do saque não será zero:

```
saldo = int( input("Digite o saldo bancário") )
saque = int( input("Digite o valor de saque") )
valor_sacado = 900 + saque

if saque > 0 and saldo >= saque and valor_sacado <= 1000:
    # instruções para liberar o saque
```

7.9 Operador or

O `or` é outro operador lógico e em tradução para o português significa `ou`. Com esse operador o escopo da condição será executado quando qualquer uma das comparações resultar em verdadeiro.

No exemplo, a seguir vamos representar uma situação que ocorre nos caixas bancários que é a limitação do tipo de valor de saque que pode ser solicitado. Por exemplo, em um caixa com notas de R$ 10, R$ 20 ou R$ 50, os seguintes valores de saque serão possíveis: 10, 40, 100, mas R$ 15 não seria. Para garantir que o valor solicitado é válido, vamos utilizar o operador de resto de divisão (%) em relação às notas disponíveis. Se não houver resto para qualquer uma delas, significa que o valor pode ser entregue pela máquina:

```
if saque % 10 == 0 or saque % 20 == 0 or saque % 50 == 0:
    # escopo do saque
```

Observe como as comparações são separadas pelo operador `or`. No exemplo, basta que uma das comparações resulte em verdade para

que o escopo seja executado. Por outro lado, se todas forem falsas, não haverá a execução do escopo.

7.10 Combinando operadores and e or

Por fim, para encerrar este tema de operadores lógicos, podemos representar situações ainda mais complexas combinando os operadores `and` e `or` em uma mesma condição. O entendimento do código se torna um pouco mais complexo e sugerimos reler esta seção caso tenha dúvidas.

No código a seguir nós unificamos as últimas comparações apresentadas em uma única, veja o resultado:

```
saldo = int( input("Digite o saldo bancário") )
saque = int( input("Digite o valor de saque") )
valor_sacado = 900 + saque

if (saque%10 == 0 or saque%20 == 0 or saque%50 == 0) and (saque > 0 and
saldo >= saque and valor_sacado <= 1000):
    # escopo do saque
```

O código apresentado restringe o saque para valores que sejam compatíveis com as notas do caixa bancário **e** obedeça às múltiplas restrições de saque. Podemos dividir a instrução que define isto em duas partes:

Parte 1	Parte 2
saque % 10 == 0 or saque % 20 == 0 or saque % 50 == 0	saque > 0 and saldo >= saque and valor_sacado <= 1000

Perceba que o operador `and` é utilizado para conectar a parte 1 e 2, de modo que a execução do escopo depende que as comparações

de ambas as partes sejam verdadeiras. Vale lembrar que para a primeira parte basta que uma das comparações resulte em verdade.

Questões

1) Qual o papel dos operadores lógicos em uma condição?

2) Qual a diferença entre o operador `and` e `or`?

3) Qual será a resposta do algoritmo a seguir?

```
saque = 100
saldo = 30
valorSacado = 90
limiteDiario = 200

if saque <= saldo and valorSacado+saque <= limiteDiario:
    print("Saque realizado")
elif saque >= saldo or valorSacado+saque > limiteDiario :
    print("Saque não pode ser realizado")
else:
    print("Nenhuma das ações será executada")
```

a) Saque realizado

b) Saque não pode ser realizado

c) Nenhuma das ações será executada

Respostas

1) Qual o papel dos operadores lógicos em uma condição?

R: Os operadores lógicos possibilitam a inclusão de múltiplas verificações em uma única condição. Eles são úteis para representar condições mais complexas onde apenas uma verificação não seria o suficiente.

2) Qual a diferença entre o operador `and` e `or`?

R: O `and` assegura que o escopo de uma condição somente será executado quando todas as situações em análise resultarem em verdade. Por sua vez, o `or` é utilizando quando desejamos no mínimo uma situação de verdade.

3) Qual será a resposta do algoritmo a seguir?

```
saque = 100
saldo = 30
valorSacado = 90
limiteDiario = 200

if saque <= saldo and valorSacado+saque <= limiteDiario:
    print("Saque realizado")
elif saque >= saldo or valorSacado+saque > limiteDiario :
    print("Saque não pode ser realizado")
else:
    print("Nenhuma das ações será executada")
```

a) Saque realizado

b) Saque não pode ser realizado

c) Nenhuma das ações será executada

R: Letra b.

7.11 Condições complexas com elif

A instrução `else` e os operadores `and` e `or` possibilitaram a elaboração de condições mais complexas. No entanto, apenas com o uso destes recursos temos uma limitação para representar determinadas situações. Por exemplo, na tabela a seguir, temos três cenários que podem acontecer em um caixa eletrônico de um banco. Perceba como esses cenários possuem alguma relação entre si, pois

quando um deles não é verdadeiro, devemos analisar o próximo e assim em diante.

Cenário	Critérios	Ação de resposta
Pode realizar o saque	1. Possui saldo na conta; 2. Não realizou mais que R$ 1.000 de saques no dia; 3. O valor do saque somado ao que já foi sacado no dia não ultrapassa o limite anterior.	Liberar o dinheiro; Apresentar uma mensagem: "Você realizou um saque com sucesso."
Não possui saldo na conta	1. O valor de saque é maior que o saldo bancário.	Apresentar uma mensagem: "Você não possui saldo suficiente para realizar essa operação."
Excedeu limite de saques diários	1. Possui saldo na conta; 2. Já sacou mais que R$ 1.000 no dia.	Apresentar uma mensagem: "Você excedeu o seu limite diário de saques."

Com o uso do `if` e `else` nós conseguimos criar uma comparação e oferecer uma resposta quando esta for verdadeira ou falsa. Entretanto, os cenários apresentados ilustram situações mais complexas e formadas por mais de uma comparação. Por exemplo, mesmo que o cliente não possa realizar o saque (o primeiro cenário é falso), há mais de uma causa para este problema. Assim, precisamos realizar outras análises para definir o que aconteceu.

Percebe-se que o `if` e `else` são limitados para representar a situação acima. Nestes casos, precisaremos de mais uma instrução conhecida por `elif`. Ela permite incluir outras comparações à nossa

estrutura condicional e apresenta uma sintaxe similar ao `if`, onde é preciso incluir uma comparação entre dados:

```
if comparação:
    # escopo do if
elif comparação:
    # escopo do elif
else:
    # escopo do else
```

Assim como o `else`, não é possível utilizar o `elif` sem um `if`. Além disso, o conceito de comparação excludente continua valendo. Ou seja, quando um `if` ou `elif` for verdadeiro, os demais não serão executados. Da mesma forma que, se nenhum deles for verdadeiro e houver um `else`, então seu escopo será executado.

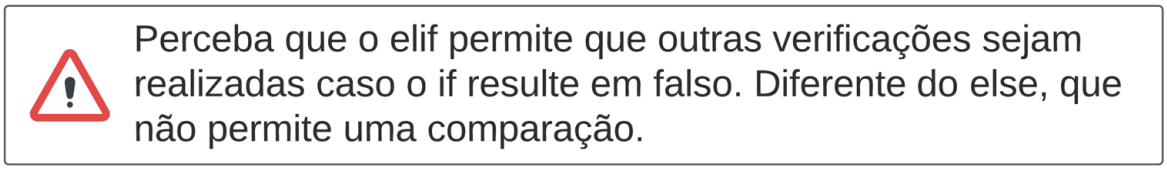

Figura 7.7: Perceba que o elif permite que outras verificações sejam realizadas caso o if resulte em falso. Diferente do else, que não permite uma comparação.

A seguir apresentamos o uso do `if`, `elif` e `else` em uma única estrutura condicional. A ideia é construir um menu de operações de um caixa bancário, onde uma variável nomeada `operacao` armazena a ação desejada pelo cliente, que pode ser saque ou depósito. Caso ele escreva uma diferente dessas, então receberá uma mensagem de aviso:

```
operacao = input()
if operacao == "saque":
    # escopo do saque
elif operacao == "depósito":
    # escopo do depósito
```

```
else:
    print("Operação não existente.")
```

O próximo exemplo é o mais complexo apresentado até o momento, pois combina uma estrutura condicional composta por `if`, `elif` e `else`, com o uso dos operadores lógicos `and` e `or`. O algoritmo a seguir ilustra a implementação dos cenários bancários apresentados anteriormente:

```
saldo = int( input("Digite o saldo bancário") )
saque = int( input("Digite o valor de saque") )
valor_sacado = 900 + saque


if saque <= saldo and valor_sacado <= 1000:
    print("Você realizou um saque com sucesso.")
elif saque <= saldo and valor_sacado > 1000:
    print("Você excedeu o seu limite diário de saques.")
else:
    print("Você não possui saldo suficiente para realizar essa operação.")
```

Para entender o algoritmo vamos dividi-los em partes:

```
saldo = int( input("Digite o saldo bancário") )
saque = int( input("Digite o valor de saque") )
valor_sacado = 900 + saque
```

Nesse trecho estamos criando as variáveis que vão guardar os valores de saque e saldo. Também estamos considerando que o cliente já sacou R$ 900 no dia e que este montante precisa ser somado ao valor solicitado para o saque. Essa informação é necessária para verificar se a soma excede o limite de saque estabelecido em R$ 1.000.

```
if saque <= saldo and valor_sacado <= 1000:
    print("Você realizou um saque com sucesso.")
```

Nesta comparação realiza-se duas verificações: a) se há dinheiro na conta suficiente para o saque e b) se o limite diário ainda não foi alcançado.

```
elif saque <= saldo and valor_sacado > 1000:
    print("Você excedeu o seu limite diário de saques.")
```

A segunda comparação será executada caso o `if` resulte em falso. Nela, verifica-se o caso onde há saldo, mas houve excesso no valor diário de saque.

```
else:
    print("Você não possui saldo suficiente para realizar essa operação.")
```

Por fim, quando as situações acima resultam em falso, por exclusão podemos assumir que o problema é a falta de saldo em sua conta bancária.

Questões

1) Qual a sintaxe da instrução `elif`?

2) Qual a diferença entre as instruções `elif` e `else`?

3) O algoritmo a seguir identifica o tipo de um Pokémon a partir de uma entrada. Assim, se a entrada for Pikachu, então deve-se apresentar o texto "elétrico"; para o Squirtle, apresenta-se "água" e para o Bulbassauro será exibido o texto "grama". No entanto o código apresenta alguns problemas. Faça a sua correção.

```
pokemon = input("Digite o nome do Pokémon")

if pokemon == "Pikachu":
    print("elétrico")
else pokemon == "Squirtle":
    print("água")
elif:
    print("grama")
```

4) Crie um algoritmo que vai ler um número inteiro do teclado e indicar se ele é positivo, negativo ou igual a zero. Essa informação deve ser apresentada na tela.

Respostas

1) Qual a sintaxe da instrução `elif` ?

```
elif << comparação >>:
    # escopo do elif
```

2) Qual a diferença entre as instruções `elif` e `else` ?

R: O `elif` permite que uma comparação seja realizada, enquanto que o `else` não. Além disso, a execução do `else` ocorrerá quando nenhuma das comparações de uma estrutura condicional forem verdadeiras.

3) O algoritmo a seguir identifica o tipo de um Pokémon a partir de uma entrada. Assim, se a entrada for Pikachu, então deve-se apresentar o texto "elétrico"; para o Squirtle, apresenta-se "água" e para o Bulbassauro será exibido o texto "grama". No entanto o código apresenta alguns problemas. Faça a sua correção.

```
pokemon = input("Digite o nome do Pokémon")

if pokemon == "Pikachu":
    print("elétrico")
elif pokemon == "Squirtle":
    print("água")
else:
    print("grama")
```

Comentário: não deve existir a comparação do `else` , que deveria estar no `elif` .

4) Crie um algoritmo que vai ler um número inteiro do teclado e indicar se ele é positivo, negativo ou igual a zero. Essa informação deve ser apresentada na tela.

```
numero = int( input() )

if numero > 0:
    print("positivo")
elif numero < 0:
    print("negativo")
```

```
else:
    print("zero")
```

7.12 Operador de expressão condicional

É possível simplificar a escrita de condições `if` e `else` por meio de um operador conhecido por *expressão condicional*. Esse operador também conhecido em outras linguagens de programação pelo nome *operador ternário*.

Apresentamos a seguir dois códigos, o primeiro com o uso do `else` e no segundo realizamos a reescrita deste (ou como chamamos na programação: refatoramos) para utilizar uma expressão condicional:

Código original

```
numero = int( input("Digite um número") )

if numero > 4:
    print("Número é maior que 4")
else:
    print("Número não é maior que 4")
```

Refatoração:

```
numero = int( input("Digite um número") )
print("Numero é maior que 4") if numero > 4 else print("Numero não é maior que 4")
```

A sintaxe para uso de expressões condicionais segue o seguinte formato:

[resposta se verdadeiro] if [comparação] else [resposta se falso]

O uso de expressões condicionais torna o seu código compacto. No entanto, esse recurso se aplica somente em situações onde os

escopos apresentam apenas uma instrução.

7.13 Erros mais comuns

Os códigos a seguir exemplificam os principais erros com condições:

Ausência de dois pontos. O sinal de `:` não é utilizado para iniciar o escopo de uma condição (`if` , `else` ou `elif`).

Código com problema:

```
nome = input("Digite um nome")
if nome == "Leonardo"
```

Código corrigido:

```
nome = input("Digite um nome")
if nome == "Leonardo":
```

Falta de indentação. Os códigos que fazem parte do escopo de uma condição precisam estar indentados (espaçados):

Código com problema:

```
nome = input("Digite um nome")
if nome == "Leonardo":
print("Olá Leonardo, como vai?")
```

Código corrigido:

```
nome = input("Digite um nome")
if nome == "Leonardo":
    print("Olá Leonardo, como vai?")
```

Apenas uma igualdade na comparação. A comparação entre dados deve ser feita com dois sinais de igualdade. Muitos aprendizes utilizam apenas um.

Código com problema:

```
nome = input("Digite um nome")
if nome = "Leonardo":
```

Código corrigido:

```
nome = input("Digite um nome")
if nome == "Leonardo":
```

Apenas um dado na comparação. Cada comparação realizada deve utilizar um par de dados, com exceção de variáveis booleanas. Esse problema é mais visualizado quando se utilizam os operadores `or` e `and`.

Código com problema:

```
nome = input("Digite um nome")
if nome == "Leonardo" or "Gabriel":
    # escopo da condição
```

Código corrigido:

```
nome = input("Digite um nome")
if nome == "Leonardo" or nome == "Gabriel":
```

Operadores lógicos escritos em maiúsculo. Os operadores `or` e `and` devem ser escritos em minúsculo.

Código com problema:

```
nome = input("Digite um nome")
if nome == "Leonardo" OR nome == "Gabriel":
```

Código corrigido:

```
nome = input("Digite um nome")
if nome == "Leonardo" or nome == "Gabriel":
```

Condição sem comparação. As instruções `if` e `elif` demandam alguma comparação.

Código com problema:

```
nome = input("Digite um nome")
if nome == "Leonardo":
    # escopo do if
elif:
    # escopo do elif
```

Código corrigido:

```
nome = input("Digite um nome")
if nome == "Leonardo":
    # escopo do if
elif nome == "Gabriel":
    # escopo do elif
```

Comparação no else. O `else` não pode apresentar uma comparação.

Código com problema:

```
nome = input("Digite um nome")
if nome == "Leonardo":
    # escopo do if
else nome == "Gabriel":
    # escopo do elif
```

Código corrigido:

```
nome = input("Digite um nome")
if nome == "Leonardo":
    # escopo do if
else:
    # escopo do elif
```

7.14 Resumo e mapa mental

- Há situações onde a execução de um conjunto de instruções depende de algum critério;
- As instruções de condições são a forma utilizada para fazer as verificações necessárias que vão determinar quando um conjunto de instruções serão executadas;
- Ao conjunto de instruções que pertencem a uma condição damos o nome de escopo;
- Há três instruções que são utilizadas para representar as condições: `if`, `elif` e `else`.
- A comparação de uma condição é realizada por meio de operadores matemáticos.
- O uso de operadores lógicos permite realizar mais de uma comparação em uma condição. São exemplos o `and` e o `or`.

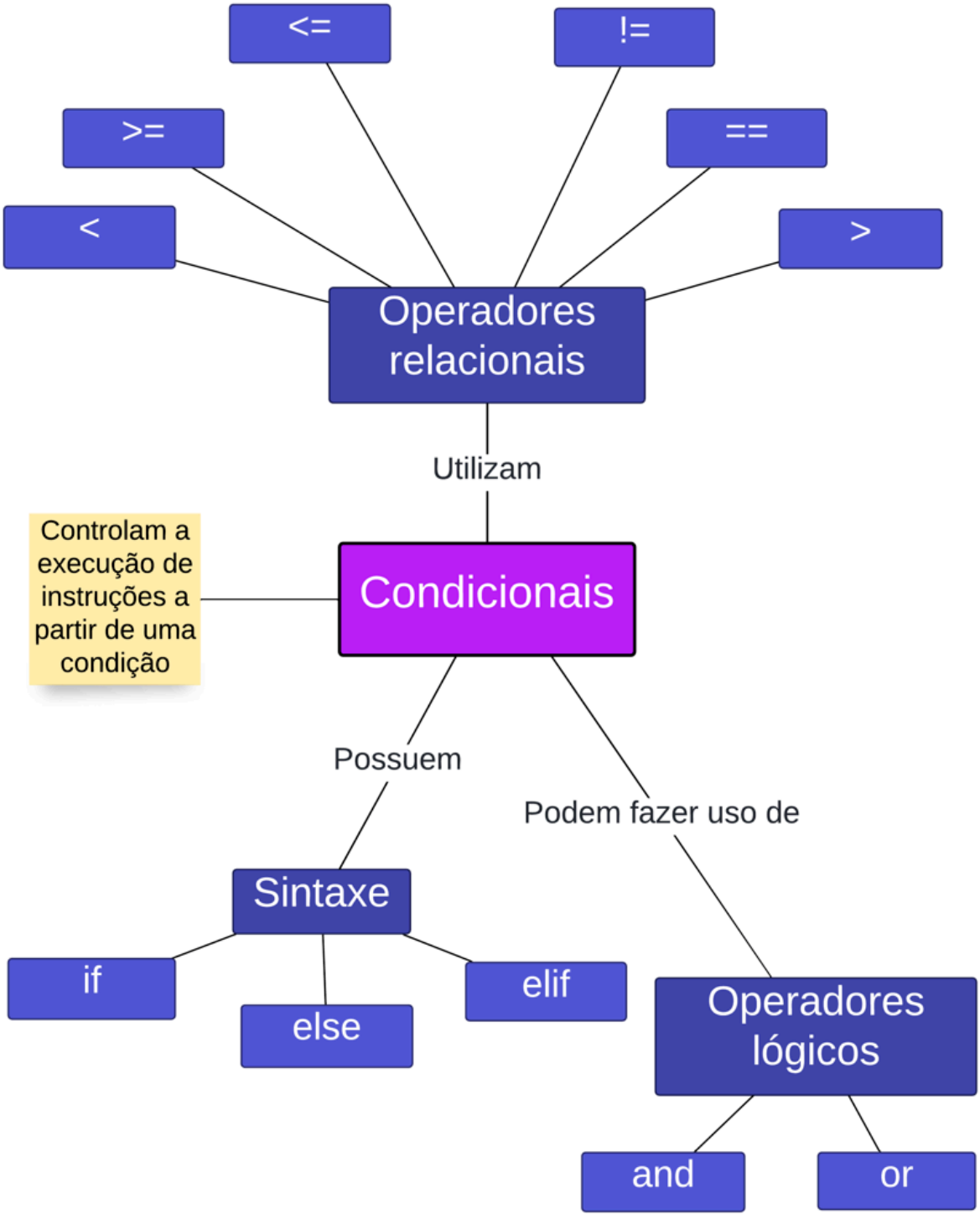
<=
!=
>=
==
<
>
Operadores relacionais
Utilizam
Controlam a execução de instruções a partir de uma condição
Condicionais
Possuem
Podem fazer uso de
Sintaxe
if
else
elif
Operadores lógicos
and
or

Figura 7.8: Mapa mental do capítulo 7.

CAPÍTULO 8
Repetição de instruções

1. O que você já sabe?

A programação é formada por um conjunto de recursos e cada um possui um propósito bem definido. As variáveis são usadas para armazenar dados e as condições permitem controlar quando um conjunto específico de instruções devem ser executadas.

2. O que veremos?

À medida que nossos programas se tornam mais complexos, precisamos de mais recursos para resolver determinados problemas. Por exemplo, imagine desenvolver um algoritmo para calcular a média final de 300 alunos. Com os conhecimentos apresentados neste livro, certamente o código será extenso, pois serão trezentas instruções para leitura de dados, trezentas fórmulas de cálculo, entre outras operações que se repetem. Consequentemente, ele também será difícil de modificar e pouco legível.

Em situações como essa, que envolve a repetição de instruções, podemos simplificá-las por meio de um recurso conhecido por **repetições**.

3. Por que é importante?

Repetir a execução de instruções de programação é uma situação comum. No entanto, apresentamos alguns dos problemas que ocorrem, como o aumento do código e menor legibilidade. Podemos mitigá-los utilizando repetições, um recurso essencial para qualquer aplicação.

O que você vai aprender?

Ao término deste capítulo você compreenderá:

1. A importância das estruturas de repetição para a programação;
2. Os diferentes tipos de repetição que o Python oferece;
3. A sintaxe para o uso de repetições na programação;
4. Formas mais complexas de repetição, como repetições aninhadas.

8.1 Quando utilizar repetições?

Nossos softwares frequentemente precisam repetir a execução de um conjunto de instruções. Ilustramos essa situação com o exemplo de cálculo de notas apresentado no início deste capítulo. As repetições também estão presentes em nossas vidas, pois diariamente realizamos muitas ações que se repetem. Por exemplo, após escolhermos os produtos de uma feira no supermercado e nos dirigirmos ao caixa, precisamos retirá-los do carrinho, um a um, e colocá-los na esteira, até o carrinho esvaziar. Nesse caso há duas ações que se repetem: 1) retirar o produto do carrinho; e 2) colocá-lo na esteira do caixa. Da mesma forma, um funcionário do supermercado também realizará ações repetidas para concluir a sua compra, como passar cada produto por um leitor de códigos de barras e em seguida embalá-lo. Esses procedimentos se repetem até não existirem mais produtos.

O software que controla as vendas do supermercado também vai repetir a execução de um conjunto de instruções múltiplas vezes. Ele precisa estar preparado para receber a leitura do código do produto, carregar suas informações e somar seu valor ao total da compra. Essas ações se repetem para cada produto. Podemos pensar no seguinte algoritmo para representar a situação que foi descrita:

- Ler o código de barras do produto;
- Obter o valor do produto;
- Somar esse valor ao total da compra;
- Ler o código de barras do produto;
- Obter o valor do produto
- Somar esse valor ao total da compra;
- … repetir as ações acima para cada produto
- Apresentar o valor total da compra.

Preste atenção nesse algoritmo, pois vamos transformá-lo em código ao longo deste capítulo. Começaremos pela identificação do problema que será resolvido: calcular o valor total de uma venda do supermercado. Em seguida, podemos identificar as entradas e saídas desse algoritmo. Observe que todas as ações dependem da identificação de qual produto está sendo comprado, neste caso, definido pelo seu código de barras. Para simplificar, não utilizaremos um leitor de códigos de barras tradicional, mas sim o comando `input` . Por fim, como o cliente deseja visualizar o valor total de sua compra, esta é a saída.

Vamos representar os conceitos do parágrafo anterior em um código Python. Para simplificar, vamos considerar que existem apenas três produtos: feijão, arroz e macarrão.

```
valor_total_compra = 0
valor_feijao = 5.5
valor_arroz = 4
valor_macarrao = 2

produto = input("Digite o código do produto")

print(valor_total_compra)
```

Nesse código, criamos a variável `valor_total_compra` para armazenar o valor total da compra, que inicia em zero. Também foram criadas três variáveis para armazenar os valores dos produtos. Por fim,

criamos uma instrução para ler o código do produto escolhido pelo cliente e outra para apresentar o valor da compra ao cliente.

No entanto, após a leitura do código do produto é preciso identificá-lo. Esse procedimento é necessário para definir qual valor será somado ao total da compra. Utilizaremos uma condição, como visualizado a seguir:

```
valor_total_compra = 0
valor_feijao = 5.5
valor_arroz = 4
valor_macarrao = 2

produto = input("Digite o código do produto")

if produto == "feijão":
    valor_total_compra += valor_feijao
elif produto == "arroz":
    valor_total_compra += valor_arroz
elif produto == "macarrão":
    valor_total_compra += valor_macarrao

print(valor_total_compra)
```

Apesar de funcionar, nosso algoritmo possui uma limitação, pois ele processa a venda de apenas um produto. Como dificilmente uma compra possui apenas essa quantidade, vamos melhorar o algoritmo para poder atender a três produtos diferentes:

```
valor_total_compra = 0
valor_feijao = 5.5
valor_arroz = 4
valor_macarrao = 2

produto = input("Digite o código do primeiro produto")

if produto == "feijão":
    valor_total_compra += valor_feijao
elif produto == "arroz":
```

```
    valor_total_compra += valor_arroz
elif produto == "macarrão":
    valor_total_compra += valor_macarrao

produto_dois = input("Digite o código do segundo produto")

if produto_dois == "feijão":
    valor_total_compra += valor_feijao
elif produto_dois == "arroz":
    valor_total_compra += valor_arroz
elif produto_dois == "macarrão":
    valor_total_compra += valor_macarrao

produto_tres = input("Digite o código do terceiro produto")

if produto_tres == "feijão":
    valor_total_compra += valor_feijao
elif produto_tres == "arroz":
    valor_total_compra += valor_arroz
elif produto_tres == "macarrão":
    valor_total_compra += valor_macarrao

print(valor_total_compra)
```

Observe como ele cresceu de tamanho. Consegue visualizar algum problema nesse código? Ele está funcional, **mas há o uso repetido de instruções**. Primeiramente, a legibilidade do código se torna difícil, pois há muitas instruções para compreender. Um outro problema, ainda mais grave, é que modificações no código são difíceis de serem realizadas. Por exemplo, e se o cálculo do preço de venda dos produtos for alterado? Teríamos de modificar todas as instruções onde isso é realizado, o que é trabalhoso. Pior, se você esquecer de alterar algum dos cálculos, o seu software apresentará um erro lógico, pois calculará o valor total da compra de forma incorreta. Esses problemas se multiplicam se pensarmos em aumentar o total de produtos vendidos.

Os problemas apresentados anteriormente indicam que há algo de errado em nosso código e que ele pode ser melhorado. Este é um conceito chamado na computação por *"coding smells"*. Para resolver situações como a exemplificada aqui utilizaremos um recurso da programação conhecido por **repetições ou loop**.

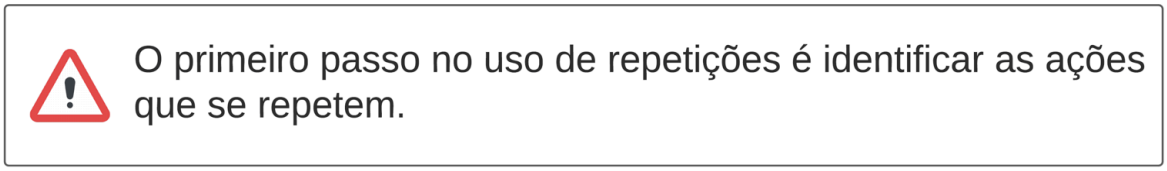

Figura 8.1: O primeiro passo no uso de repetições é encontrar quais ações se repetem.

8.2 Repetições na programação

Para criar repetições no Python utilizaremos duas instruções: `for` e `while`. Ambas possuem um funcionamento similar, executando múltiplas vezes um bloco de códigos. O que muda entre elas é a forma como definimos quantas vezes uma repetição deve ocorrer.

Vamos iniciar pela instrução `for`. No algoritmo a seguir faremos a impressão de um texto cem vezes. Não se preocupe com os detalhes, pois os explicaremos na sequência:

```
for contador in range(100):
    print("Olá mundo")
```

O `for` utilizado na primeira linha deste código indica que uma repetição será realizada. O quantitativo de vezes é definido na instrução `range(100)`. Por enquanto ignore o trecho `contador in`, pois não o usaremos neste exemplo, mas é preciso informá-lo de acordo com a sintaxe do Python. Na sequência, temos a instrução que será repetida, neste caso um simples `print`. Observe que, assim como as condições, as repetições possuem um escopo, e por isso é preciso

utilizar um espaçamento em suas instruções. Vale lembrar que não há um limite no número de instruções que fazem parte do escopo.

No exemplo anterior, ao encontrar a instrução `for`, o computador saberá que deve executar o escopo da repetição por 100 vezes. Para isso, internamente, o computador mantém uma contagem sobre quantas repetições já foram realizadas. O Python disponibiliza essa informação para nós por meio da variável `contador`, que pode ser útil caso queira realizar alguma ação quando alcançar um determinado número de vezes. Poderíamos dar qualquer outro nome para essa variável, mas a sintaxe precisa ser respeitada:

```
for variavel-contadora in range( quantidade-repeticoes ):
```

Para compreender o funcionamento da variável contadora veja o seguinte exemplo:

```
for contador in range(100):
    print(contador)
```

Ao executar esse algoritmo, a seguinte saída será produzida: `0 1 2 … 98 99`. Observe como a variável `contador` é iniciada em 0 e finaliza em 100-1.

As repetições possuem um escopo e suas instruções devem estar indentadas.

Figura 8.2: As repetições possuem um escopo e suas instruções devem estar indentadas.

É importante entender como o computador faz o processamento das repetições, pois isso ajudará na compreensão desse conceito. A execução do escopo da repetição ocorre de cima para baixo até a sua última instrução. Após isso, o computador destrói os dados

criados no escopo e retorna para a primeira instrução do escopo. Antes de executá-lo mais uma vez, ele verifica se a repetição deve ocorrer, ou seja, não chegou ao limite estabelecido no `range`. Vamos apresentar um código para que essas informações sejam mais facilmente entendidas. Vamos utilizar uma variável para apresentar um texto, juntamente do número de repetições que foram realizadas.

```
for contador in range(100):
    texto = "Repetição de número "+str(contador)
    print(texto)
```

Observe que dentro do escopo da repetição temos duas instruções. A primeira realiza a concatenação entre um texto com o dado guardado na variável `contador`. Como este é um número, precisamos convertê-lo para *String*. Na sequência, o dado guardado em `texto` é apresentado ao usuário. O código resultará na saída:

```
Repetição de número 0
Repetição de número 1
...
Repetição de número 99
```

Você pode estar pensando que o computador está guardando na memória 100 variáveis textuais. No entanto, cada vez que a repetição chega ao fim do escopo, a variável `texto` é destruída. Ela é recriada quando há uma nova iteração inicia-se (ou ciclo de repetição, nome utilizado para indicar uma execução do escopo), sendo mais uma vez destruída ao término do escopo. Esse processo se repete enquanto o loop for executado. Após não existirem mais repetições o fluxo de execução do algoritmo segue normalmente.

Antes de avançar para um exemplo mais complexo é preciso entender como nosso algoritmo trabalha com instruções que se repetem e outras que não. Veja o exemplo a seguir:

```
print("Início do algoritmo")
for contador in range(100):
    print(contador)
```

```
print("Fim do algoritmo")
```

A execução do código anterior apresentará na tela as seguintes informações:

```
Início do algoritmo
0
1
...
99
Fim do algoritmo
```

Perceba que a execução das instruções na primeira e última linha ocorre por apenas uma vez, pois elas estão fora do escopo da repetição.

Vamos utilizar instruções de repetição para transformar o algoritmo de supermercado apresentado no início do capítulo. O primeiro passo no uso desse recurso é identificar as instruções se repetem, pois elas serão isoladas no escopo do loop. Pelo exemplo do início do capítulo sabemos que as instruções repetidas são:

```
produto = input("Digite o código do produto")

if produto == "feijão":
    valor_total_compra += valor_feijao
elif produto == "arroz":
    valor_total_compra += valor_arroz
elif produto == "macarrão":
    valor_total_compra += valor_macarrao
```

Em seguida, devemos avaliar por quantas vezes queremos que as instruções se repitam. Chamaremos isto de **condição de parada**. Em nosso exemplo foi definido que a repetição será executada três vezes. Vamos ao exemplo completo:

```
valor_total_compra = 0
valor_feijao = 5.5
valor_arroz = 4
```

```
valor_macarrao = 2

for contador in range(3):
    produto = input("Digite o código do produto")

    if produto == "feijão":
        valor_total_compra += valor_feijao
    elif produto == "arroz":
        valor_total_compra += valor_arroz
    elif produto == "macarrão":
        valor_total_compra += valor_macarrao

print(valor_total_compra)
```

Perceba como nosso algoritmo reduziu de tamanho em comparação ao exemplo sem repetições. Além disso, qualquer alteração no escopo da repetição valerá para todas as suas execuções.

Um ponto para ficar atento neste exemplo é que temos dois escopos: um para a repetição e outro para as condições. Assim, observe que o `if` e `elif` estão indentados em relação à repetição e que as instruções pertencentes às condições possuem outra indentação.

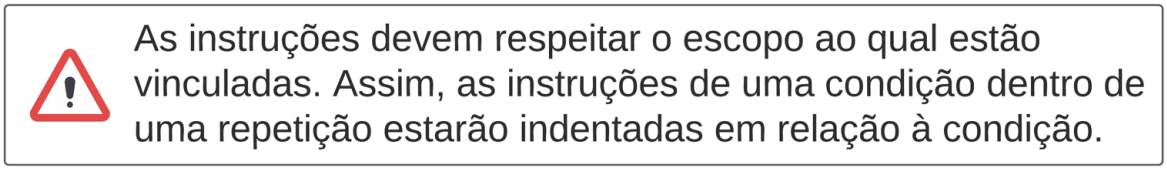

Figura 8.3: As instruções devem respeitar o escopo ao qual estão vinculadas. Assim, as instruções de uma condição dentro de uma repetição estarão indentadas em relação à condição.

Iteração por uma String

Por fim, o `for` também pode ser utilizado para iterar por uma *String*. Isso é útil, pois possibilita percorrer cada uma de suas letras, como

pode ser visto no exemplo a seguir:

```
texto = "Olá mundo!"
for letra in texto:
    print(letra)
```

A saída deste algoritmo será:

```
O
l
á

m
u
n
d
o
!
```

Qualquer recurso que possa ser percorrido com uma repetição é conhecido por `iterator`. Até o momento você conheceu dois, o `range` e a *String*.

Questão

1) Quais as instruções que são utilizadas para criar uma repetição no Python?

2) Qual o problema em não utilizar repetição?

3) Identifique no algoritmo a seguir as instruções que poderiam fazer parte de um escopo de repetição:

```
nome_um = input("Digite o primeiro nome")
nome_dois = input("Digite o segundo nome")
nome_tres = input("Digite o terceiro nome")
nome_quatro = input("Digite o quarto nome")

nomes = nome_um+" "+nome_dois+" "+nome_tres+" "+nome_quatro
```

```
print(f"Olá {nomes}")
```

Respostas

1) Quais as instruções que são utilizadas para criar uma repetição no Python?

R: `while` e `for`.

2) Qual o problema em não utilizar repetição?

R: Será preciso reescrever múltiplas vezes instruções similares, o que afeta a legibilidade do algoritmo. Além disso, torna-se mais difícil a realização de modificações nesses trechos de código, pois seria necessário atualizar todas as partes repetidas.

3) Identifique no algoritmo a seguir as instruções que poderiam fazer parte de um escopo de repetição:

```
nome_um = input("Digite o primeiro nome")
nome_dois = input("Digite o segundo nome")
nome_tres = input("Digite o terceiro nome")
nome_quatro = input("Digite o quarto nome")

nomes = nome_um+" "+nome_dois+" "+nome_tres+" "+nome_quatro

print(f"Olá {nomes}")
```

R:

```
nome_um = input("Digite o primeiro nome")
nome_dois = input("Digite o segundo nome")
nome_tres = input("Digite o terceiro nome")
nome_quatro = input("Digite o quarto nome")
```

Código com repetição:

```
nomes = ""

for contagem in range(4):
```

```
    nome = input("Digite um nome")
    nomes += nome+" "

print(f"Olá {nomes}")
```

8.3 Instrução range

A instrução `range` foi utilizada para obter uma sequência de números. Por exemplo, `range(100)` produzirá 100 números, iniciando em zero e encerrando em 100-1. Essa sequência é utilizada pelo `for` para determinar quantas vezes seu escopo será executado, que nesse caso será igual ao tamanho da sequência.

Há outras formas de utilizar o range e para isso é preciso entender a sua sintaxe:

```
range(início, fim, incremento)
```

Os três dados que `range` recebe vão modificar a forma como a sequência de números é gerada. O primeiro determina em qual número queremos iniciar, seguido pelo número em que se deve parar e, por fim, o incremento entre cada um dos números.

Quando escrevemos `range(100)` não informamos todos os dados mencionados. O Python permite essa situação, pois entende que seria o equivalente à instrução: `range(0, 100, 1)`.

Vamos ver alguns exemplos para entender as diferentes sequências geradas pelo `range`:

Exemplo de uso do range	Sequência numérica criada
range(5)	0, 1, 2, 3 e 4
range(2,5)	2, 3 e 4

Exemplo de uso do range	Sequência numérica criada
range(1, 10, 2)	1, 3, 5, 7 e 9

Observe que a sequência gerada irá até o número informado para o fim e subtraído de um. Além disso, o incremento faz com que os números variem dentro da sequência e consequentemente influencia o total de números gerados. Por fim, quando utilizamos apenas dois dados, como no exemplo (2,5), o Python entende que o incremento deve ser de um em um.

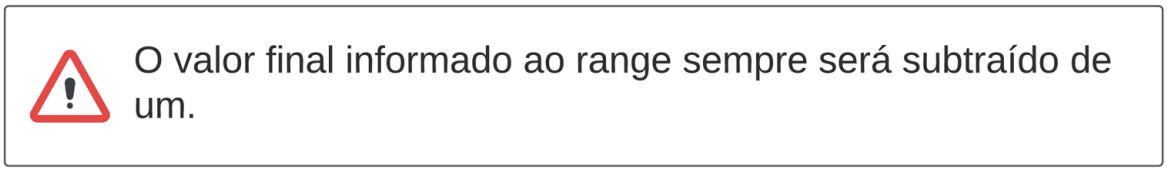

Figura 8.4: O valor final informado ao range sempre será subtraído de um.

Questão

1) Qual a sintaxe da instrução `range` e o que são as informações que recebe?

2) Qual o papel da função `range` ?

3) Qual a saída do algoritmo a seguir:

```
for x in range(2,10,2):
    print(x)
```

4) Construa um algoritmo e utilize as instruções `for` e `range` para realizar uma repetição. A repetição deverá executar por 10 vezes e imprimir apenas os valores ímpares entre 0 a 9.

Respostas

1) Qual a sintaxe da instrução `range` e o que são as informações que recebe?

R: Sua sintaxe é: `range(inicio, fim, incremento)`, onde `inicio` determina o valor inicial do intervalo, `fim` é o valor final e `incremento` determina a variação entre os números.

2) Qual o papel da função `range`?

R: A instrução `range` é utilizada para construir uma sequência numérica dentro de um intervalo definido. Por exemplo, a chamada `range(2,10,2)` resultará em uma sequência com os valores 2, 4, 6 e 8. Essa sequência pode ser utilizada no `for` para determinar a quantidade de repetições que serão realizadas.

3) Qual a saída do algoritmo a seguir:

```
for x in range(2,10,2):
    print(x)
```

R: A saída do algoritmo será: `2 4 6 8`

4) Construa um algoritmo e utilize as instruções `for` e `range` para realizar uma repetição. A repetição deverá executar por 10 vezes e imprimir apenas os valores ímpares entre 0 a 9.

```
for x in range(10):
    if x%2 != 0:
        print(x)
```

8.4 Repetição com while

O `while` representa outra instrução para criação de repetições no Python. Em tradução para o português, essa palavra significa *enquanto*. É importante você compreender esse sentido, pois ajudará no entendimento desta instrução.

O funcionamento do `while` é simples, ele testará uma condição que, sendo verdadeira, vai disparar a execução do escopo de repetição. Sua sintaxe é representada por:

```
while comparação:
    # escopo da repetição
```

Para ajudar nesse entendimento veja um exemplo:

```
contador = 1

while contador < 10:
    contador +=1
    print(contador)
```

O algoritmo testa a condição `contador < 10`. Como o valor dessa variável inicia-se em 1, o escopo da repetição será executado. Consequentemente, o valor da variável *contador* será incrementado para dois e esse dado será impresso na tela. Após o término do escopo, o `while` testará a condição mais uma vez, que também resultará em verdadeiro, pois 2 é menor que 10. Os procedimentos descritos se repetem até o momento em que a comparação passa a ser falsa.

Vamos explicar uma situação em que a pessoa programadora deve ter atenção ao uso do `while`. Observe o algoritmo a seguir:

```
texto = "Olá mundo"

while texto == "Olá mundo":
    print(texto)
```

Você consegue prever o que será impresso na tela ao executar esse algoritmo? Basicamente, nós teremos uma sequência de vários `"Olá mundo"` sendo impressos infinitamente, evento conhecido por *loop infinito*, o que não é desejado. Por que isso aconteceu? O uso do `while` requer que a condição testada em algum momento resulte em falso. No exemplo, a comparação realizada sempre será verdadeira, pois a variável `texto` não tem o seu valor alterado.

Assim, é fundamental que dentro do escopo de sua repetição as variáveis testadas na comparação tenham seus valores alterados para não ocasionar o problema descrito. Vamos ver uma forma de corrigir o problema apresentado:

```
texto = "Olá mundo"

while texto == "Olá mundo":
    print(texto)
    texto = input()
```

Observe que dentro do escopo da repetição incluímos uma leitura de texto com `input`, o que pode ocasionar a mudança na variável `texto`. Se o usuário digitar um texto diferente de `Olá mundo`, a repetição será encerrada.

A decisão sobre quando utilizar o `for` ou o `while` é simples. Quando a quantidade de repetições for conhecida previamente, utilizamos o `for` e quando não sabemos dessa informação utiliza-se o `while`. Vamos exemplificar, lembram do algoritmo do supermercado? Nós limitamos a 3 produtos em cada venda, no entanto, no mundo real não é dessa forma, pois não sabemos quantos produtos um cliente deseja comprar. Isso é um forte indicativo de que deveríamos utilizar o `while` para resolver esse problema.

A seguir, apresenta-se o exemplo do supermercado escrito com `while`. Dessa forma, permitimos que seja vendida uma quantidade indeterminada de produtos.

```
valor_total_compra = 0
valor_feijao = 5.5
valor_arroz = 4
valor_macarrao = 2
produto = ""

while produto != "sair":

    if produto == "feijão":
        valor_total_compra += valor_feijao
```

```
    elif produto == "arroz":
        valor_total_compra += valor_arroz
    elif produto == "macarrão":
        valor_total_compra += valor_macarrao

    produto = input("Digite o código do produto")


print(valor_total_compra)
```

No algoritmo há algumas mudanças significativas em comparação ao exemplo com `for`. Primeiramente foi preciso declarar a variável `produto` antes do `while` para que ela pudesse ser utilizada na comparação. Seu valor foi inicializado como uma *String* vazia para que a repetição iniciasse, uma vez que esse valor é diferente de `sair`. Enquanto o operador do supermercado não digitar `sair`, a repetição ocorrerá e os produtos poderão ser incluídos na venda.

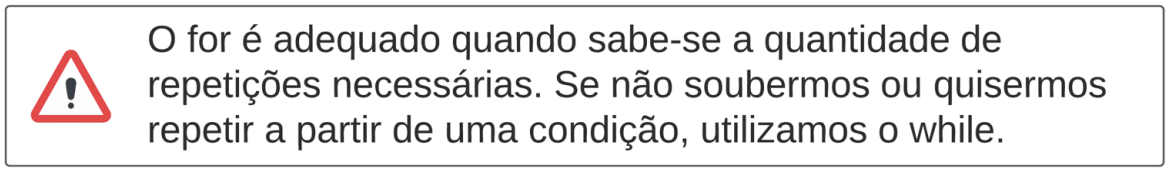

Figura 8.5: O for é utilizado quando sabemos a quantidade de repetições que serão realizadas. Se não soubermos o ou quisermos utilizar uma comparação, o while é recomendado.

Questão

1) Qual a sintaxe da instrução `while`?

2) Qual é a saída do algoritmo a seguir?

```
contador = 0
while contador < 10:
    if contador % 2 == 0:
```

```
        print(contador)
    contador += 1
```

3) Quais as diferenças entre o uso das instruções `while` e `for`?

Respostas

1) Qual a sintaxe da instrução `while`?

```
while comparação:
    # escopo
```

2) Qual é a saída do algoritmo a seguir?

```
contador = 0
while contador < 10:
    if contador % 2 == 0:
        print(contador)
    contador += 1
```

R: `0 2 4 6 8`

3) Quais as diferenças entre o uso das instruções `while` e `for`?

R: A instrução `for` realiza a repetição a partir de iteradores, que podem ser produzidos com a instrução `range`, a partir de *Strings*. Assim, sabe-se *a priori* a quantidade de repetições que serão produzidas. Por outro lado, o `while` utiliza uma condição para determinar se a repetição deve ser realizada. Dessa forma, mantendo-se o resultado dessa comparação em verdadeiro, pode-se executar a repetição por uma quantidade indeterminada de vezes.

8.5 Instruções break e continue

As instruções `break` e `continue` permitem ao programador um maior controle sobre a quantidade de repetições que são executadas.

Para entender o funcionamento da instrução `break`, considere a situação onde desejamos imprimir apenas o primeiro número ímpar de uma sequência de números, ignorando a repetição dos demais. Vamos analisar essa situação. Não interessa a execução da repetição após encontrarmos o primeiro número ímpar, portanto, é preciso pensar em uma estratégia para interromper a execução da repetição. É nesse contexto que o `break` é utilizado, pois ele permite executar uma repetição antes da condição de parada ser estabelecida. Veja a sua aplicação a seguir:

```
for contador in range(10):
    if contador % 2 != 0: # Essa condição é verdadeira para números
ímpares
        print(contador)
        break
```

Esse recurso é útil, pois a execução de cada ciclo de repetição apresenta um custo de processamento ao computador. Ainda que na maioria dos casos o(a) programador(a) não precise se preocupar com isso, se for possível evitar a execução de ciclos desnecessários, a aplicação apresentará uma melhor performance.

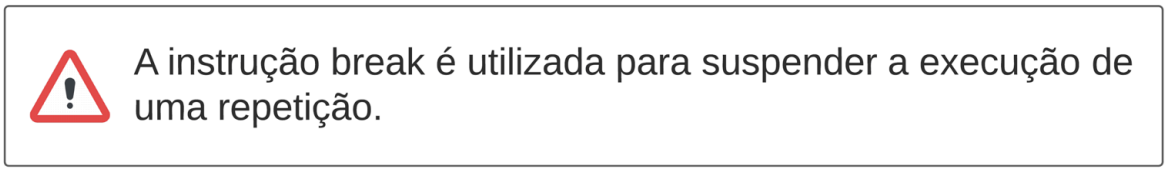

Figura 8.6: A instrução break é utilizada para suspender a execução de uma repetição.

O `continue` é uma instrução que possibilita ignorar um determinado ciclo da repetição, mas diferente da instrução `break` os demais ciclos não são encerrados. Ele é útil quando não desejamos realizada nada em um ciclo específico. Veja o algoritmo a seguir, que realiza a impressão de um conjunto de números, com exceção do 1 e 8.

```
for x in range(10):
    if x == 1 or x == 8:
        continue
    print(x)
```

Figura 8.7: A instrução continue permite avançar para o próximo ciclo da repetição.

As instruções `break` e `continue` podem ser utilizadas com o `for` ou `while`.

Questões

1) Qual o nome da instrução que realiza a interrupção de um ciclo de repetição, sem afetar os demais?

2) Qual o nome da instrução que interrompe todos os ciclos de repetição subsequentes?

3) Considere o algoritmo a seguir, qual será a sua saída?

```
for x in range(10):
    if x == 2:
        continue
    if x == 5:
        break
    print(x)
```

4) Considere o seguinte problema: um número de 0 a 9 será lido do teclado. Na sequência, você deverá implementar uma repetição com 10 ciclos e imprimir o número do contador. Quando o contador alcançar o número digitado pelo usuário, deve-se parar a execução e

não nos interessa mais repetir os demais ciclos. Qual seria o algoritmo para este problema?

Respostas

1) Qual o nome da instrução que realiza a interrupção de um ciclo de repetição, sem afetar os demais?

R: `continue`

2) Qual o nome da instrução que interrompe todos os ciclos de repetição subsequentes?

R: `break`

3) Considere o algoritmo a seguir, qual será a sua saída?

```
for x in range(10):
    if x == 2:
        continue
    if x == 5:
        break
    print(x)
```

R: `0 1 3 4`

4) Considere o seguinte problema: um número de 0 a 9 será lido do teclado. Na sequência, você deverá implementar uma repetição com 10 ciclos e imprimir o número do contador. Quando o contador alcançar o número digitado pelo usuário, deve-se parar a execução e não nos interessa mais repetir os demais ciclos. Qual seria o algoritmo para este problema?

```
contador = 0
numero = int(input("Digite um número inteiro de 0 a 9"))
while contador < 10:
    if contador == numero:
       break
    print(contador)
    contador += 1
```

8.6 Repetições aninhadas

Vimos que é possível utilizar condições dentro de repetições e vice-versa. Uma outra possibilidade é utilizar repetições dentro do escopo de outras repetições. Esse conceito é conhecido por **repetição aninhada**. Para compreender o funcionamento de uma repetição aninhada veja o algoritmo a seguir e tente descobrir a saída correta.

```
contador = 0
while contador < 3:
    outro_contador = -1
    while outro_contador < 2:
        print(outro_contador)
        outro_contador += 1
    contador +=1
    print(contador)
```

Qual a saída produzida pelo algoritmo?

a) 0 -1 0 1 1 -1 0 1 2 -1 0 1

b) -1 0 1 1 -1 0 1 2 -1 0 1 3

c) -1 0 0 1 1 2

Se você respondeu letra b) acertou!

Em uma repetição aninhada, o Python executa o escopo da primeira repetição e, ao encontrar a instrução de início da segunda repetição, vai executar o seu escopo e repetir esse procedimento até a sua condição de parada. Quando isso ocorrer, o escopo da primeira repetição volta a ser executado e esse procedimento se repete até que a condição de parada da primeira repetição seja alcançada.

Também é possível utilizar repetições aninhadas mesclando `for` com `while` e vice-versa:

```
for x in range(5):
    contador = 0
    while contador < 3:
        print(contador)
        contador += 1
```

A execução desse código apresentará a saída:

```
0
1
2
0
1
2
0
1
2
0
1
2
0
1
2
```

Questão

1) O que são repetições aninhadas?

2) Qual será a resposta do algoritmo a seguir:

```
contador = 2
for x in range(2):
    print(x)
    while contador > 0:
        print(contador)
        contador -= 1
```

3) Crie um algoritmo com duas repetições, sendo uma delas aninhada. Na repetição primária serão impressos na tela apenas os

valores pares, enquanto que, na segunda, serão impressos na tela os valores ímpares. Cada algoritmo deve ser repetido 5 vezes.

Respostas

1) O que são repetições aninhadas?

R: É uma forma de utilizar repetições dentro do escopo de outra repetição.

2) Qual será a resposta do algoritmo a seguir:

```
contador = 2
for x in range(2):
    print(x)
    while contador > 0:
        print(contador)
        contador -= 1
```

R: `0 2 1 1`

4) Crie um algoritmo com duas repetições, sendo uma delas aninhada. Na repetição primária, serão impressos na tela apenas os valores pares, enquanto que. na segunda, serão impressos na tela os valores ímpares. Cada algoritmo deve ser repetido 5 vezes.

```
for x in range(5):
    if x%2 == 0:
        print(x)
    for y in range(5):
        if y%2 != 0:
            print(y)
```

8.7 Cláusula else em repetições

No capítulo 7 apresentamos as estruturas condicionais que são formadas por `if`, `elif` e `else`. Este último possibilita criar um

conjunto de instruções de resposta para quando as comparações com `if` ou `elif` não resultarem em verdade. Também é possível utilizar o `else` com repetições, mas ele apresenta um propósito diferente a depender do tipo de repetição utilizada. Vamos explicar com exemplos.

Muitas vezes utilizamos o `for` para verificar se um determinado dado existe em um conjunto de informações. Por exemplo, saber se há algum número ímpar nos dados produzidos por um `range` . Vamos ver a implementação dessa funcionalidade:

```
for numero in range(2, 5, 2):
    if numero %2 != 0:
        print("Número ímpar")
        break
```

Observe que após a identificação do número ímpar suspendemos a execução do restante do código com um `break` , pois não interessa continuar a repetição se a informação já foi localizada. O código está funcional, mas e para exibir uma mensagem quando não houver um número ímpar? Nesse caso pode-se criar uma variável para dizer se o número foi ou não encontrado:

```
possui_numero_impar = False

for numero in range(2, 5, 2):
    if numero %2 != 0:
        possui_numero_impar = True
        print("Número ímpar")
        break

if possui_numero_impar == False:
    print("Não tem número ímpar")
```

Criamos uma variável `possui_numero_impar` , que é inicializada em False. Seu valor é alterado dentro do escopo da repetição quando houver um número ímpar. Posteriormente, uma condição é

implementada após a repetição e utiliza essa variável para exibir ou não uma mensagem.

Apesar de funcional, o código pode ser melhorado utilizando recursos do próprio Python, como o `else`. Seu escopo somente será executado **SE** a repetição não tiver sido interrompida por um `break`. A seguir, apresentamos o código anterior modificado com o uso do `else`:

```
for numero in range(2, 5, 2):
    if numero %2 != 0:
        print("Número ímpar")
        break
else:
    print("Não tem número ímpar")
```

Caso exista um número ímpar, então o `break` suspenderá a execução do *loop* e assim o escopo do `else` não é executado. Em caso contrário, quando não há números ímpares, após o término da repetição o escopo de `else` é executado. Observe que o código apresenta menos linha e não faz mais uso de uma variável para controlar se existem números ímpares.

Outra possibilidade de uso do `else` é combiná-lo com uma repetição `while`. Neste caso, o escopo do `else` é executado quando a comparação realizada no `while` resultar em falso. O código a seguir demonstra o uso desse recurso. Solicitamos ao usuário que digite uma letra, enquanto não for uma vogal o código vai repetir. Ao encontrar uma vogal, o escopo do `else` é executado.

```
letra = input("Digite uma letra")
while letra != "a" and letra != "e" and letra != "i" and letra != "o" and
letra != "u":
    letra = input("Digite uma letra")
else:
    print("Vogal encontrada")
```

O escopo do `else` sempre é executado quando a comparação do `while` resultar em falso.

8.8 Erros mais comuns

Ausência de dois pontos. Não escrever o sinal de `:` (dois pontos) é um dos principais erros cometidos na hora de escrever uma repetição.

Código com problema:

```
for numero in range(5)
```

Código corrigido:

```
for numero in range(5):
```

Apenas uma igualdade em uma repetição com while. Este problema acontece quando apenas um sinal de igualdade é utilizado em uma comparação com `while`.

Código com problema:

```
nome = "Leonardo"
while nome = "Leonardo":
    # escopo do while
```

Código corrigido:

```
nome = "Leonardo"
while nome == "Leonardo":
    # escopo do while
```

Ausência do operador in. A iteração em uma *String*, `range`, entre outros iteráveis requer o uso do operador `in`.

Código com problema:

```
for numero range(3):
```

Código corrigido:

```
for numero in range(3):
```

Apenas um dado na comparação. Este erro também foi apresentado no capítulo de condições. Ele ocorre quando apenas um dado é utilizado em uma comparação do `while`.

Código com problema:

```
nome = input("Digite um nome")
while nome == "Leonardo" or "Gabriel":
```

Código corrigido:

```
nome = input("Digite um nome")
while nome == "Leonardo" or nome == "Gabriel":
```

Não criar uma condição de início verdadeira para uma repetição com while. O escopo do `while` somente será executado quando a condição testada resultar em verdade. Caso isso nunca ocorra, consequentemente não haverá a execução das suas instruções.

Código com problema:

```
contador = 2

while contador == 0:
```

Código corrigido:

```
contador = 0

while contador == 0:
```

Não modificar o dado comparado no while. Em cada iteração do `while`, a condição é analisada para determinar se a repetição deve ser executada. Se o dado utilizado na comparação não for

alterado dentro do escopo da repetição, pode-se criar uma situação de loop infinito.

Código com problema:

```
numero = 0

while numero == 0:
    print(numero)
```

Código corrigido:

```
numero = 0

while numero == 0:
    print(numero)
    numero = int( input("Digite um número") )
```

8.9 Resumo e mapa mental

- Neste capítulo nós aprendemos o conceito de repetição e como esse recurso possibilita repetir a execução de instruções de programação;
- Há diversos benefícios, como a redução no tamanho do código e facilidade para realização de mudanças;
- Há duas instruções de repetição no Python: `for` e `while`;
- O uso do `for` é adequado para situações onde sabemos previamente quantas vezes desejamos repetir um código;
- A instrução `while` é adequada para repetir instruções por uma quantidade de vezes indefinida;
- As instruções `break` e `continue` podem ser utilizadas nas repetições para suspender a repetição de todo um loop ou apenas de um ciclo, respectivamente;
- Por fim, é possível incluir instruções de repetições dentro do escopo de outra repetição, conceito conhecido por repetição aninhada.

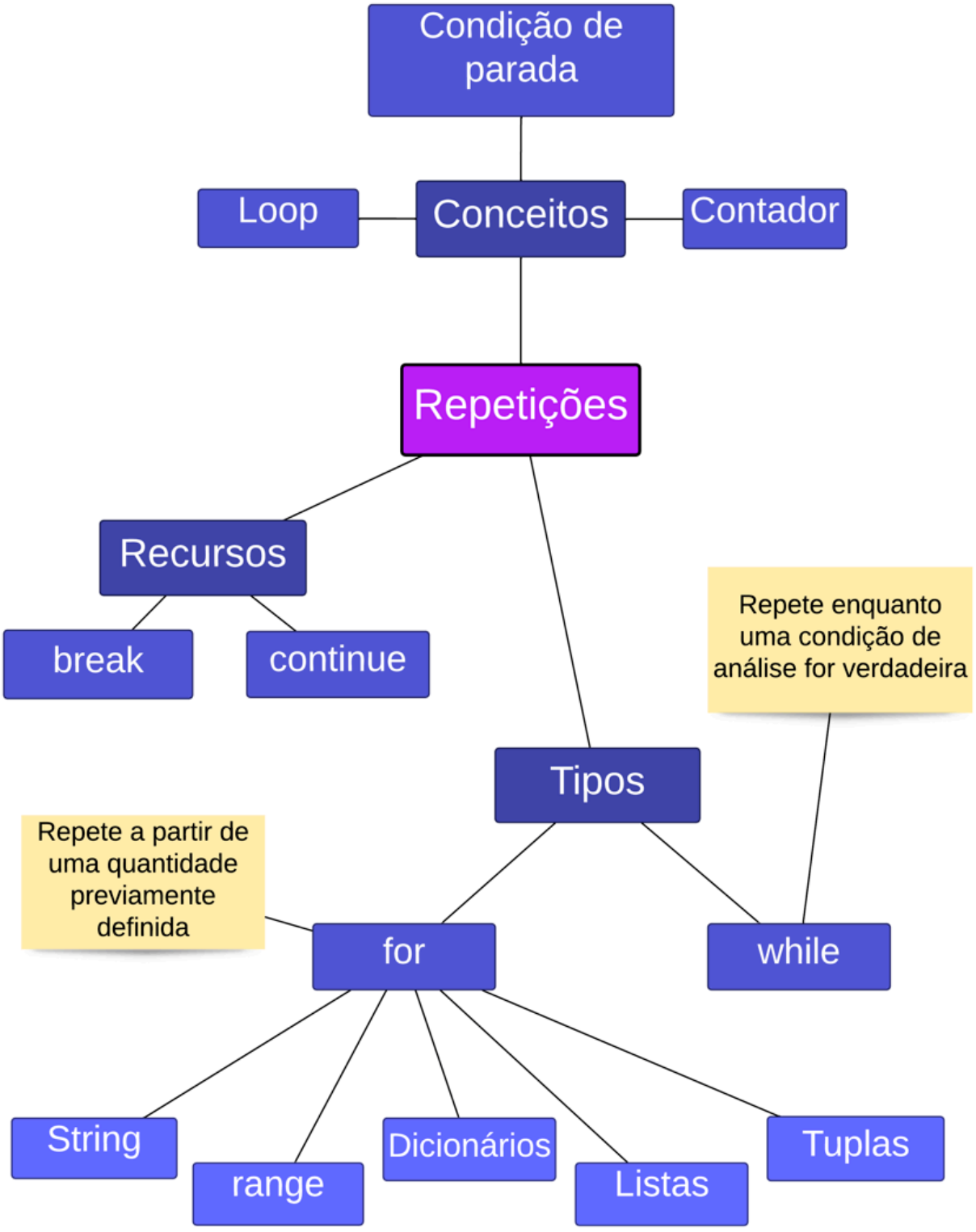
Condição de parada
Loop
Conceitos
Contador
Repetições
Recursos
break
continue
Repete enquanto uma condição de análise for verdadeira
Tipos
Repete a partir de uma quantidade previamente definida
for
while
String
range
Dicionários
Listas
Tuplas

Figura 8.8: Mapa mental do capítulo 8.

Organizando o código com funções

Capítulo 9
Funções no Python

1. O que você já sabe?

Em um programa guardamos dados (variáveis), controlamos quais instruções serão executadas a partir de certos critérios (condições) e podemos repetir a execução de instruções por meio de repetições (`for` e `while`). Também existem outros recursos, como o `print` e `input` , que são conhecidos por **funções**.

2. O que veremos?

As funções são utilizadas com frequência na programação. Você mesmo já fez uso delas, por meio de algumas que o Python oferece, como o `print` , `input` e `range` . Neste capítulo vamos conhecer outras funcionalidades que estão disponíveis nesta linguagem de programação, como também criar as nossas próprias.

3. Por que é importante?

O `input` possibilitou ao nosso programa ler dados do teclado. Para isso, não foi preciso conhecer os mecanismos de comunicação entre este periférico e o sistema operacional, pois a função fez isso para nós com apenas uma linha de código. Existem outras funcionalidades oferecidas pelo Python e que podemos utilizar em nossos softwares, facilitando nosso trabalho como programadores. Você também poderá criar as suas próprias funções e logo aprenderá os benefícios em fazer isto.

O que você vai aprender?

Ao término deste capítulo você compreenderá:

1. O que são funções na programação;
2. Situações que justificam o uso de funções;
3. A sintaxe para o uso e criação deste recurso no Python;
4. Algumas das funções que o Python nos oferece.

9.1 Qual problema as funções resolvem

Podemos fazer uma analogia entre uma função com a prestação de serviço realizada por um profissional. Por exemplo, quando você contrata um pintor ou pintora, essa pessoa utilizará todo o seu conhecimento e realizar as ações necessárias para entregar um cômodo pintado. Você não precisa conhecer os detalhes de como o serviço é realizado e também não terá de realizar esse trabalho.

Como falamos anteriormente, no caso da função `input`, quem a criou precisou conhecer o processo de comunicação do teclado (hardware) com o sistema operacional do computador (software), processar essa informação e entregar para nós (programadores e programadoras) o que foi digitado. Para realizar tudo isso foi preciso escrever dezenas ou mesmo centenas de instruções. No entanto, nós, programadores, podemos utilizar esse recurso com apenas uma linha de código. Além disso, uma função pode ser reutilizada quantas vezes forem necessárias. Este conceito é denominado *reúso* e marca uma das principais características desse recurso. Consegue imaginar o trabalho que o Python nos poupou?

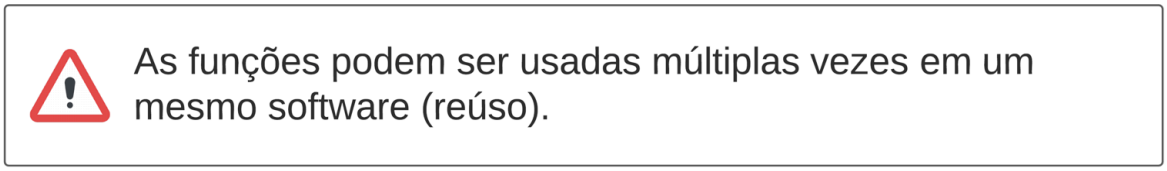

Figura 9.1: As funções podem ser usadas múltiplas vezes em um mesmo software (reúso).

Uma função entrega alguma funcionalidade ao nosso software. Por exemplo, o `print` permite a impressão de informações na tela e o `input`, a leitura de dados do teclado. Há diversas outras funções oferecidas pelo Python, como também é possível criar suas próprias.

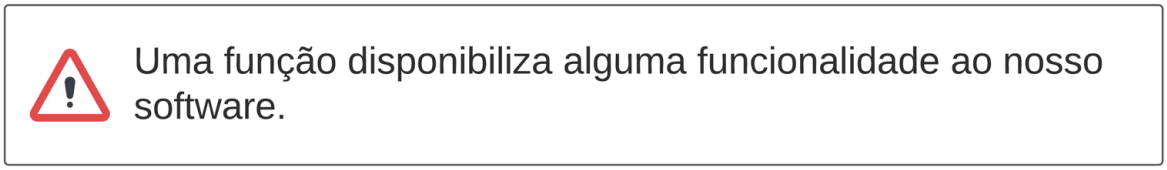

Figura 9.2: Uma função disponibiliza alguma funcionalidade ao nosso software.

Questões

1) Cite 3 funções do Python que você já utilizou.

2) Quais os principais benefícios do uso das funções em nossos programas?

Respostas

1) Cite 3 funções do Python que você já utilizou.

R: `input`, `print` e `int`.

2) Quais os principais benefícios do uso das funções em nossos programas?

R: Funções possibilitam a realização de uma ação que em muitos casos não precisa ser criada pela pessoa programadora. Nestes casos, ela apenas faz uso das funcionalidades sem se preocupar em como elas foram criadas.

Internamente uma função, em geral, é formada por um conjunto de instruções. Quando uma função é utilizada, essas instruções são executadas, mas com o benefício de que para o programador isso é feito com apenas uma linha de código. Essa possibilidade de encapsular um conjunto de códigos é um outro benefício das funções.

9.2 Sintaxe para execução

As funções dividem-se em execução (também conhecido por uso ou chamada) e criação. Neste primeiro momento vamos abordar a execução, pois é um procedimento simples e já realizado por você com algumas funções do próprio Python.

Certamente você reconhecerá o código a seguir:

```
print("Olá mundo")
```

Nesse código há uma execução da função de nome `print`. Vamos analisá-la em partes. O primeiro ponto a ser observado é que **todas as funções precisam de um nome** e isso é necessário, pois, sem ele, como diremos ao Python qual função executar? As funções também possuem parênteses de abertura e fechamento e que em alguns casos podem vir acompanhados de um ou mais dados informados dentro deles. Ao fazer isso, estamos enviando dados para "dentro" da função, que poderá utilizá-los para alguma finalidade. No exemplo anterior escrevemos uma *String* que será processada pelo `print` e apresentada no monitor do usuário. Falaremos mais sobre este conceito conhecido por **parâmetro** em breve.

Algumas funções podem ainda receber mais de um parâmetro, como o próprio `print`. Veja no exemplo a seguir:

```
print("Olá mundo", 123)
```

Observe que os parâmetros são separados por vírgulas e a execução da instrução acima apresentará na tela: `Olá mundo 123`

Você também reconhecerá a sintaxe a seguir, que é um pouco diferente da anterior:

```
nome = input()
```

Você consegue perceber a diferença? Vamos explicar, observe que na mesma linha onde executamos o `input` há também a criação de uma variável `nome`. Isso ocorre, pois a execução dessa função cria para nós um dado (o conteúdo digitado pelo usuário em seu teclado) e que precisa ser armazenado em alguma variável. Nesse caso, dizemos que a função **retornou uma informação**. Esse conceito também será explorado nas próximas seções.

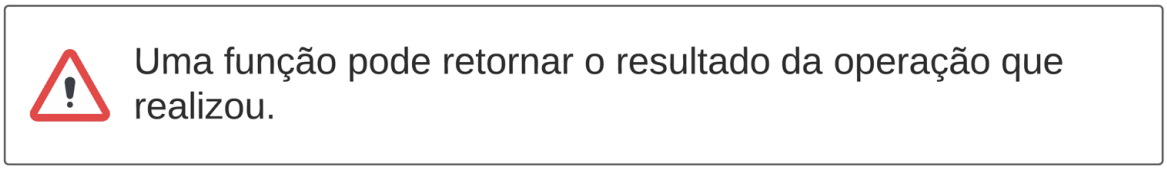

Figura 9.3: Uma função pode retornar o resultado da operação que realizou.

Percebe-se que a sintaxe para execução de uma função vai depender dos seus parâmetros e de haver ou não o retorno de alguma informação. Nos exemplos, `print` não apresenta retorno, mas recebe parâmetros, enquanto na execução do `input` não utilizamos nenhum parâmetro, mas havia um retorno. Assim, podemos resumir as sintaxes para execução das funções da seguinte forma.

Quando não há retorno:

```
nome-da-funcao( parâmetros )
```

Funções que possuem retorno:

```
variavel-para-armazenar-retorno = nome-da-funcao( parâmetros )
```

Onde `nome-da-funcao` é o nome da função que será executada, seguido obrigatoriamente dos parênteses e dos parâmetros, quando houver, separados por vírgulas se houver mais de um. Caso a função produza um retorno, essa informação deverá ser armazenada em uma variável, nomeada `variavel-para-armazenar-retorno`.

Questões

1) Quais as partes de uma função?

2) Qual a sintaxe para a execução de uma função?

Respostas

1) Quais as partes de uma função?

R: Nome, parâmetros e retorno.

2) Qual a sintaxe para a execução de uma função?

R: A sintaxe vai depender da estrutura da função. Deve-se informar o seu nome e os parâmetros, se houver, são escritos entre os parênteses e separados por vírgulas. Caso exista um retorno, deve-se ter a declaração de uma variável juntamente com a execução da função.

```
nome-da-variavel = nome-da-funcao( parâmetros )
```

9.3 Funções prontas para uso do Python

Como sabemos, o Python dispõe de uma variedade de funções prontas para os mais diversos usos. É importante que você as conheça, pois isso otimiza seu tempo de desenvolvimento (já pensou o tempo investido para criar o seu próprio *input*? Seria realmente necessário?). Além disso, temos a garantia de que são funções escritas com qualidade.

Nesta seção serão apresentadas algumas funções e no site oficial (https://docs.python.org/) você encontrará a relação completa.

Funções para manipular strings

Com frequência precisaremos realizar operações em nossas *Strings*, como localizar caracteres ou palavras em um texto, dividi-lo em partes, entre outros. A seguir são apresentadas algumas funções com esse propósito, seguido de exemplos para facilitar a sua compreensão.

- **count**

Contabiliza a quantidade de ocorrências de um texto/caractere em uma *String*. Sintaxe:

```
resultado = variavel-string.count( caractere/texto )
```

Exemplo:

```
texto = "olá mundo"
ocorrencias = texto.count("o")
print(ocorrencias)
```

Saída: `2`

- **find**

Localiza um texto/caractere em uma *String* e retorna a posição da primeira ocorrência, iniciando a contagem em zero. Se não localizar, então o resultado será -1. Sintaxe:

```
resultado = variavel-string.find( caractere/texto )
```

```
texto = "olá mundo"
pesquisa = texto.find("m")
print(pesquisa)
```

Saída: `4`

- **split**

Divide uma *String* em partes a partir de um padrão que é informado como parâmetro. Ao localizar esse padrão na *String*, haverá a separação do texto próximo a ele das outras partes. Sem parâmetro, a quebra ocorrerá para cada espaço vazio encontrado. O retorno da função é uma lista (tema do nosso próximo capítulo) com cada divisão da *String*. Sintaxe:

```
resultado = variavel-string.split( padrão )
```

```
texto = "texto, entre, virgulas"
texto_dividido = texto.split(",")
print(texto_dividido)
```

Saída: `['texto', 'entre', 'vírgulas']`.

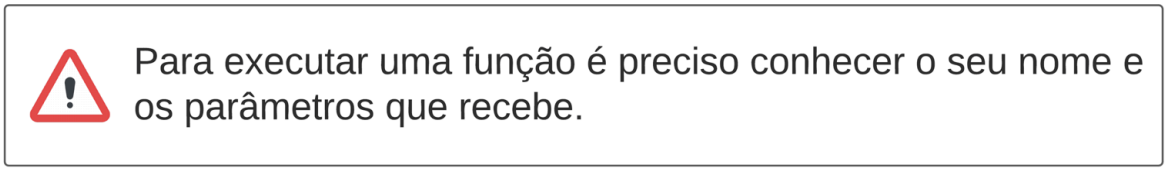

Figura 9.4: Para executar uma função é preciso conhecer o seu nome e os parâmetros que recebe.

Funções matemáticas

Algumas operações matemáticas podem ser facilmente realizadas com funções prontas do Python. Há uma pequena diferença em relação às demais funções que já utilizamos, pois é preciso escrever a instrução `import math` em nosso código. Isso é necessário, pois o Python possui uma organização modular, no qual cada módulo possui uma responsabilidade bem definida. Assim, as funções matemáticas foram organizadas em um módulo nomeado de **math**. Se tentar utilizar uma função matemática sem importar o módulo você receberá o seguinte aviso:

```
NameError: name 'math' is not defined
```

Vamos conhecer algumas das funções deste módulo.

- **ceil**

Arredonda um número decimal para o inteiro mais próximo. Sintaxe:

```
resultado = math.ceil( numero )
```

Exemplo:

```
import math
```

```
numero_arredondado = math.ceil(3.6)
print(numero_arredondado)
```

Saída: `4`

- **factorial**

Calcula o fatorial de um número inteiro. Sintaxe:

```
resultado = math.factorial( numero )
```

Exemplo:

```
import math

numero_fatorial = math.factorial(5)
print(numero_fatorial)
```

Saída: `120`

- **fabs**

Transforma um número em absoluto. Sintaxe:

```
resultado = math.fabs( numero )
```

Exemplo:

```
import math

numero_absoluto = math.fabs(-2)
print(numero_absoluto)
```

Saída: `2.0`

- **pow**

Eleva um número à potência de outro. Sintaxe:

```
resultado = math.pow( numero, potência )
```

Exemplo:

```
import math

numero_potencia = math.pow(2,2)
print(numero_potencia)
```

Saída: `4.0`

- **sqrt**.

Calcula a raiz quadrada de um número. Sintaxe:

```
resultado = math.sqrt( numero )
```

Exemplo:

```
import math

numero_raiz = math.sqrt(4)
print(numero_raiz)
```

Saída: `2.0`

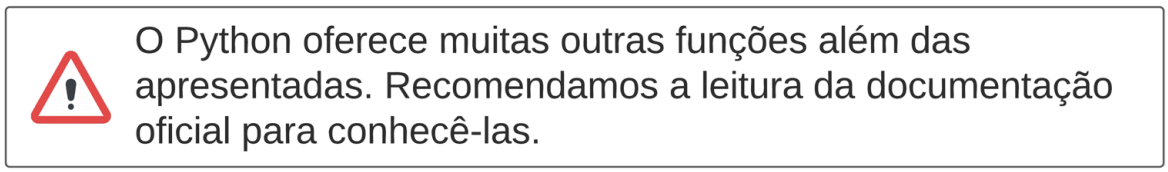

Figura 9.5: O Python oferece muitas outras funções além das apresentadas. Recomendamos a leitura da documentação oficial para conhecê-las.

Ah, você se lembra das *f-strings* que apresentamos no capítulo 6? Podemos utilizá-las para apresentar mensagens mais amigáveis para

os cálculos matemáticos:

```
import math

print(f"A raiz quadrada de quatro é: {math.sqrt(4)}")
```

O resultado será: `A raiz quadrada de quatro é: 2.0 .`

Questões

1) Qual o benefício em utilizar funções oferecidas pelo Python?

2) Assinale o nome da função que permite encontrar a posição de um determinado texto/caractere em uma *String*.

( ) count ( ) find ( ) split

3) Assinale o nome da função que divide uma *String* em partes a partir de uma condição específica.

( ) count ( ) find ( ) split

4) Quais serão as saídas do algoritmo a seguir:

```
import math

texto = "3030-132"
x = texto.split("-")
print(x)

n = math.pow(10,2)
print(n)

a = math.fabs(-5.3)
print(a)
```

5) Utilizando uma função pronta crie um algoritmo que calcule a raiz quadrada de um número.

Respostas

1) Qual o benefício em utilizar funções oferecidas pelo Python?

R: Ganha-se tempo, pois a pessoa programadora não precisa criar algoritmos para resolver determinados problemas.

2) Assinale o nome da função que permite encontrar a posição de um determinado texto/caractere em uma *String*.

( ) count (X) find ( ) split

3) Assinale o nome da função que divide uma *String* em partes a partir de uma condição específica.

( ) count ( ) find (X) split

4) Quais serão as saídas do algoritmo a seguir:

```
import math

texto = "3030-132"
x = texto.split("-")
print(x)

n = math.pow(10,2)
print(n)

a = math.fabs(-5.3)
print(a)
```

R:

```
[3030, 132]
100.0
5.3
```

5) Utilizando uma função pronta crie um algoritmo que calcule a raiz quadrada de um número.

```
import math
```

```
numero = 100
raiz = math.sqrt(numero)
```

9.4 Criação de funções

A criação de funções é uma ação que o(a) programador(a) realiza com frequência. O principal benefício é poder reutilizar um conjunto de instruções que realizam alguma ação com apenas uma linha de código. Vamos explicar...

Primeiramente é importante compreender as situações que justificam a criação de funções. Um dos indicativos são as instruções que se repetem em diferentes partes do código e com pequenas variações. Vamos analisar esse caso por meio de um algoritmo que movimenta personagens em um jogo RPG. Neste sentido, uma prática comum aos jogos eletrônicos é utilizar um plano cartesiano para representar a localização e movimentação de personagens. Você aprendeu sobre o conceito de plano ainda no ensino fundamental e ele consiste de um sistema de coordenadas dividido em dois eixos nomeados x e y. Assim, um posicionamento em um dado momento é representado por uma combinação de coordenadas em cada um dos eixos e a movimentação consiste na variação das coordenadas. Por exemplo, um personagem na posição (10, 10) que se desloca para a direita em 5 pontos terá como nova posição (15, 10).

Apresenta-se a seguir um código que realiza a movimentação em um plano cartesiano. São criadas duas coordenadas x e y, uma para cada personagem e a movimentação é feita por meio da mudança de suas coordenadas x.

```
x_personagem_principal = 10
y_personagem_principal = 10
```

```
x_personagem_ferreiro = 5
y_personagem_ferreiro = 20

# Movimenta-se o personagem principal em 5 pontos da coordenada x

x_personagem_principal += 5

# Movimenta-se o personagem ferreiro em 15 pontos da coordenada x

x_personagem_ferreiro += 15
```

Podemos tornar o código ainda mais realístico, pois após uma movimentação devemos verificar se houve uma colisão com algo que também está no mapa. Em nosso exemplo faremos essa verificação com uma edificação que está representada pelas coordenadas (15, 10). Caso isso aconteça, será apresentada uma mensagem. Para simplificar, considere que há uma colisão quando as coordenadas x e y de dois elementos estão com o mesmo valor. Vamos atualizar o algoritmo:

```
x_personagem_principal = 10
y_personagem_principal = 10

x_personagem_ferreiro = 5
y_personagem_ferreiro = 20

# Coordenadas da edificação
x_edificacao = 15
y_edificacao = 10

x_personagem_principal += 5

# Verificação de colisão para o personagem principal

if x_personagem_principal == x_edificacao and y_personagem_principal  ==
y_edificacao:
    print("Aconteceu uma colisão")

x_personagem_ferreiro += 15
```

```
# Verificação de colisão para o personagem ferreiro

if x_personagem_ferreiro == x_edificacao and y_personagem_ferreiro  ==
y_edificacao:
    print("Aconteceu uma colisão")
```

Apesar de funcional, o código apresenta um pequeno "problema". Percebe-se que as instruções de movimentação e de verificação de colisão são similares e, da forma como estão, ocasionam uma repetição em nosso código. Se houver outros personagens, a repetição se tornará ainda maior, pois teremos de escrever as mesmas instruções para eles.

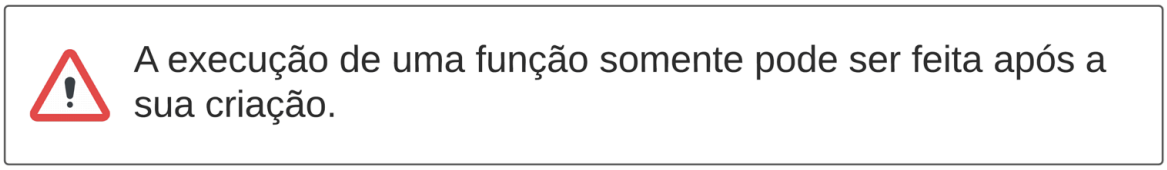

Figura 9.6: A execução de uma função somente pode ser feita após a sua criação.

Situações como a representada acima são um forte indicativo de que precisamos de uma função para encapsular o código comum às várias partes do nosso programa. Há uma série de vantagens em realizar isso, como:

1. Reduzir a repetição de códigos e facilitar a sua manutenção no longo prazo, pois se for preciso alterar algo, concentra-se em maior parte nas instruções de criação da função;
2. Também auxilia na organização do código, pois as funções reduzem a quantidade de instruções de programação e melhoram a legibilidade;
3. Favorece o reúso, pois uma função pode ser utilizada quantas vezes forem necessárias.

Preste atenção nos códigos que se repetem e observe as pequenas mudanças que existem entre eles, pois são indicativos da necessidade de uma função.

Figura 9.7: Preste atenção nos códigos que se repetem e observe as pequenas mudanças que existem entre eles, pois são indicativos da necessidade de uma função.

Sintaxe para criação de uma função

A sintaxe para a criação de uma função no Python é representada pelo seguinte código:

```
def nome-da-funcao( parâmetros-se-houver ):
     # instruções no escopo da função
    return valor-de-retorno-se-houver
```

Dividimos esse código em seis partes para facilitar sua explicação:

1. A criação de uma função inicia-se pela palavra reservada **def**.
2. Na sequência, informamos o nome da função. Em geral nós utilizamos palavras no infinitivo para indicar que uma ação será processada, exemplos: calcular, somar, processar, exibir, deletar, entre outros;
3. Escrevem-se os parênteses de abertura e fechamento. Dentro deles é preciso informar os parâmetros que a função recebe (se houver) e separá-los por vírgulas quando houver mais de um;
4. Utilizar o símbolo de `:` (dois pontos) para indicar que nas linhas seguintes será o escopo da função;
5. Escrever as instruções que fazem parte do escopo e que serão executadas quando a função for chamada. Assim como ocorreu com as condições e repetições, essas instruções precisam estar indentadas.
6. Definir o valor que será retornado utilizando a palavra `return` seguido do valor; se não houver retorno, essa instrução deve ser omitida.

Vamos ver um exemplo de criação de função nomeada `dizer_ola` que realiza uma saudação. Toda vez que ela for chamada exibirá na tela a mensagem `Olá mundo!` .

```
def dizer_ola():
    print("Olá mundo!")
```

Observe que nem todas as informações que descrevemos nos seis passos acima foram aplicadas, pois a função é bastante simples e não possui parâmetros nem retorno.

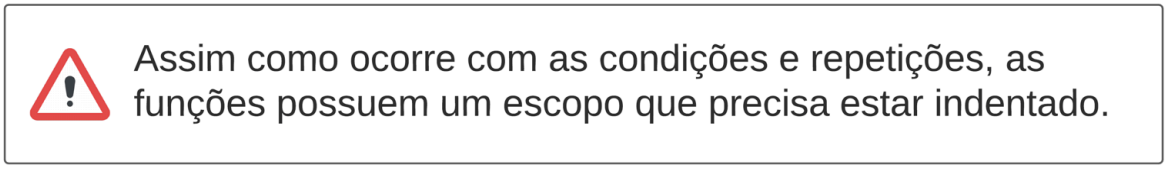

Figura 9.8: Assim como ocorre com as condições e repetições, as funções possuem um escopo que precisa estar indentado.

No entanto, apenas declarar uma função não significa que ela será executada. Como uma analogia, pense em um bolo que você fez (seria o equivalente à declaração da função), ele somente fará sentido quando alguém comê-lo (que seria o equivalente à execução da função). De forma similar, a declaração de uma função são apenas linhas que só farão sentido ao serem executadas com a chamada à função. Mas atenção, somente é possível executar uma função que já foi criada, caso contrário o Python acusará um erro: `NameError: name 'nome-da-funcao' is not defined. .`

Como exemplificado, as funções na programação dividem-se em duas partes: **declaração e chamada**. Na declaração, define-se a estrutura da função, composta pelo seu nome, parâmetros, instruções que fazem parte do seu escopo e retorno. A chamada vai executar as instruções que fazem parte do escopo da função. Assim, o código completo para o exemplo anterior seria:

```
def dizer_ola():
    print("Olá mundo!")

dizer_ola()
```

E resultará na seguinte saída: `Olá mundo!`.

As funções podem ser executadas dentro de outras. Na realidade já fizemos isso no exemplo anterior, pois observe que executamos a função `print` dentro de `dizer_ola`. Vamos apresentar um exemplo com duas funções, `dizer_ola` e `saudar`, onde uma executa a outra.

```
def dizer_ola():
    print("Olá mundo!")

def saudar():
    dizer_ola()

    print("Aqui há outro texto!")

saudar()
```

Observe que a função `dizer_ola` somente é executada dentro do escopo de `saudar`. Sendo assim, se a função `saudar` nunca for executada, consequentemente o escopo de `dizer_ola` também não será. No entanto, esse não é o caso, pois observe que na última linha estamos chamando a função `saudar`, resultando na saída:

```
Olá mundo!

Aqui há outro texto!
```

Passagem de parâmetros

Não importa quantas vezes você execute a função `dizer_ola`, a mesma resposta será produzida: a impressão do texto *Olá mundo!*. É possível tornar essa função mais interessante, adaptando-a para exibir uma mensagem personalizada. Para isso será utilizado o

conceito de parâmetros, recurso da programação que permite enviar dados para "dentro" do escopo da função.

O código anterior foi modificado para que a função receba um parâmetro. Preste atenção em como a sintaxe de declaração e execução da função são modificadas:

```
def dizer_ola( nome_usuario ):
    print("Olá", nome_usuario)

dizer_ola("Pietra")
```

A assinatura da função (termo que designa o nome da função e os parâmetros que a definem) precisou ser modificada para indicar o recebimento de um parâmetro. Na prática um parâmetro é uma variável, que no exemplo foi nomeada de `nome_usuario`. Observe agora a última linha do código onde a função é executada. Uma *String* de valor `Pietra` está sendo passada como parâmetro para a função e esse dado estará disponível no seu escopo por meio da variável `nome_usuario`, pois foi o nome que definimos. Essa variável pode ser utilizada dentro do escopo da função. Porém, fique atento, pois ela somente existe nele. O código a seguir ilustra esse problema, com o uso de um parâmetro fora do escopo da função (última linha).

```
def dizer_ola( nome_usuario ):
    print("Olá", nome_usuario)

dizer_ola("Pietra")
print(nome_usuario)
```

A execução resultará em um erro: `NameError: name 'nome_usuario' is not defined`.

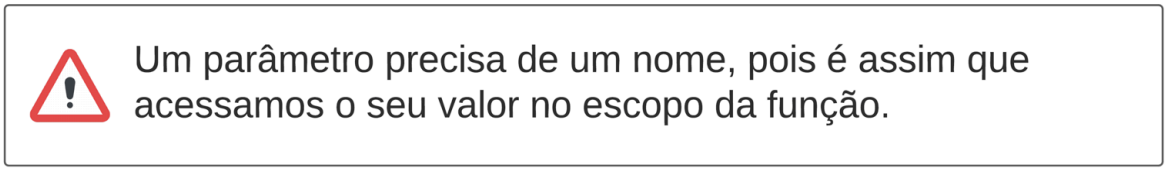

Figura 9.9: Um parâmetro precisa de um nome, pois é assim que acessamos o seu valor no escopo da função.

Mas e o que acontece se tentarmos executar a função sem parâmetros? O Python apresentará a mensagem de erro: `TypeError: dizer_ola() missing 1 required positional argument: 'nome_usuario'.`

Uma função pode receber mais de um parâmetro. Para isso, em sua assinatura deve-se separar os parâmetros com vírgula, como no exemplo a seguir:

```
def dizer_ola( nome_usuario, sobrenome):
    print("Olá", nome_usuario, sobrenome)

dizer_ola("Pietra", "Fortes")
```

Sua execução resultará na saída: `Olá Pietra Fortes .`

Observe que a vírgula é utilizada para separar os parâmetros na assinatura da função e também em sua chamada. O Python ficará encarregado de atribuir os valores aos respectivos parâmetros, seguindo a ordem em que eles foram informados. Ou seja, o dado `Pietra` ficará guardado na variável `nome_usuario` , e `Fortes` , na variável `sobrenome` . Nesse caso, dizemos que os **parâmetros são do tipo posicionais**.

Fique atento(a) à ordem em que os parâmetros são informados na execução da função.

Figura 9.10: Fique atento(a) à ordem em que os parâmetros são informados na execução da função.

Parâmetros por keyword

Também é possível indicar explicitamente para qual parâmetro queremos atribuir um determinado dado. O Python nomeia esse formato como **passagem de parâmetros por *keywords***. Há uma pequena mudança na sintaxe de execução da função, como pode ser observado a seguir:

```
nome-da-funcao( nome-do-parametro = valor )
```

Observe como entre os parênteses nós informamos qual o nome do parâmetro que receberá o valor especificado. Exemplificamos a aplicação desse recurso com o seguinte código:

```
def dizer_ola(nome_usuario, sobrenome):
    print("Olá", nome_usuario, sobrenome)

dizer_ola(sobrenome = "Fortes", nome_usuario = "Pietra")
```

A saída desse código será a mesma do anterior. No entanto, diferente dele, aqui a ordem dos parâmetros não importa. O benefício deste formato de passagem de parâmetros é a legibilidade, pois sabemos exatamente onde os dados informados serão guardados na função. Isso é especialmente útil com funções que recebem muitos parâmetros.

Parâmetros com valores padrão

Aprendemos que ao executar uma função que recebe parâmetros sem informá-los ocasionará um erro. No entanto, há uma exceção quando especificamos um valor padrão para o parâmetro. A sintaxe para realizar isso é apresentada a seguir:

```
def nome-da-funcao( nome-do-parametro = valor ):
```

Pode parecer confuso e para auxiliar na explicação utilizamos esse recurso em nossa função `dizer_ola`. A mudança pode ser observada na assinatura da função.

```
def dizer_ola(nome_usuario = "mundo"):
    print("Olá", nome_usuario)
```

Perceba que o parâmetro `nome_usuario` está recebendo o valor `mundo`. No entanto, ele somente será utilizado quando a função for chamada sem nenhum parâmetro. No código a seguir, demonstra-se a execução da função `dizer_ola` com e sem parâmetros:

```
def dizer_ola(nome_usuario = "mundo"):
    print("Olá", nome_usuario)

dizer_ola("Leonardo")
dizer_ola()
```

E produzirá a seguinte saída:

```
Olá Leonardo
Olá mundo
```

Questões

1) Qual a sintaxe para execução de uma função?

2) Qual a instrução utilizada para declarar uma função (dica: é formada por três letras)?

3) Quais as partes de uma função?

4) Qual a sintaxe para criação de uma função?

5) Qual a diferença entre a declaração de uma função e a execução de uma função?

6) O que ocorre ao tentar executar uma função que não foi declarada?

Respostas

1) Qual a sintaxe para execução de uma função?

nome-da-funcao(parâmetros)

2) Qual a instrução utilizada para declarar uma função?

R: `def`

3) Quais as partes de uma função?

R: Instrução de declaração da função def, nome, parâmetros, escopo e retorno.

4) Qual a sintaxe para criação de uma função?

```
def nome-da-funcao( parâmetros ):
    # escopo
    return
```

5) Qual a diferença entre a declaração de uma função e a execução de uma função?

R: As funções na programação se dividem em duas partes: declaração e chamada. No processo de declaração é definido o nome da função, parâmetros e instruções do escopo. A chamada de uma função é utilizada para executar seu escopo.

6) O que ocorre ao tentar executar uma função que não foi declarada?

R: O Python indicará um erro do tipo *NameError*.

9.5 Retorno de dados

Em muitos casos as funções produzem uma resposta a partir do processamento que realizam, conhecida por **retorno**. A função `input` é um exemplo e retorna o texto digitado pelo usuário. Ao executar uma função que possui um retorno é preciso indicar em qual variável esse dado será guardado, como no exemplo a seguir:

```
texto_digitado = input("Digite um texto")
```

No código, o retorno da função `input` é guardado na variável `texto_digitado`. Caso não informe uma variável o retorno da função é perdido, pois não ficaria armazenado em nenhum lugar, o que não é desejável.

É possível realizar operações no retorno de uma função. No exemplo anterior, poderíamos imprimir o que o usuário digitou na tela ou modificar esse texto:

```
texto_digitado = input("Digite um texto")
print(f"Texto que o usuário escreveu: {texto_digitado}")
```

Vamos apresentar o uso de retornos a partir da nossa função `dizer_ola`. Até o momento, essa função recebe um parâmetro, utiliza-o para concatená-lo à palavra *Olá* e em seguida imprime essa informação na tela. No entanto, há situações em que queremos realizar algum processamento no texto final antes da impressão acontecer. Para isso, a função será modificada para retornar a concatenação, em vez de imprimi-la na tela:

```
def dizer_ola( nome_usuario ):
    texto_base = "Olá "
    concatenacao = texto_base + nome_usuario
    return concatenacao
```

Consequentemente, a execução da função também vai modificar, como pode ser visto a seguir:

```
def dizer_ola( nome_usuario ):
    texto_base = "Olá "
    concatenacao = texto_base + nome_usuario
    return concatenacao

saudacao = dizer_ola("Gabriel")
```

Assim, a variável `saudacao` pode ser modificada antes de sua impressão. A seguir, apresenta-se esse caso, onde há duas execuções da função `dizer_ola` com diferentes alterações em seu retorno:

```
def dizer_ola( nome_usuario ):
    texto_base = "Olá "
    concatenacao = texto_base + nome_usuario
    return concatenacao

saudacao = dizer_ola("Gabriel")
saudacao = saudacao + "!"

print(saudacao)

saudacao_dois = dizer_ola("Leonardo")
saudacao_dois = saudacao_dois + "?"

print(saudacao_dois)
```

O algoritmo resulta na seguinte saída:

```
Ola Gabriel!
Olá Leonardo?
```

Somente é possível retornar uma informação por vez. No entanto, uma função pode possuir diferentes retornos, mas somente um será executado. Esse caso ocorre, por exemplo, quando o retorno varia a depender de uma condição. O código a seguir exemplifica essa situação com uma variação na função `dizer_ola`

que, a depender do nome informado no parâmetro, produzirá um retorno diferente.

```
def dizer_ola( nome_usuario ):
    texto_base = "Olá "
    concatenacao = texto_base + nome_usuario

    if nome_usuario == "Gabriel":
        return concatenacao + "!"
    elif nome_usuario == "Leonardo":
        return concatenacao + "?"

    return concatenacao

saudacao = dizer_ola("Gabriel")
print(saudacao)
saudacao_dois = dizer_ola("Leonardo")
print(saudacao_dois)
saudacao_tres = dizer_ola("Davi")
print(saudacao_tres)
```

Para compreender o exemplo, vamos relembrar um pouco do conceito de condições. Esse recurso permite fazer uma verificação e a partir disso definir quais instruções devem ser executadas. Na condição implementada, comparamos o valor da variável `nome_usuario` for Gabriel, então será retornado o texto *Olá Gabriel!*. Caso o valor seja Leonardo, então o retorno será *Olá Leonardo?*. Sendo um nome diferente destes dois então o retorno será *Olá nome-da-pessoa*. Assim, a execução do código apresentará a saída:

```
Olá Gabriel!
Olá Leonardo?
Olá Davi
```

Processo de criação de uma função

Exemplificaremos os conceitos apresentados até agora por meio da criação de uma função mais complexa. Para isso é preciso retornar

ao início do capítulo onde apresentamos um exemplo de movimentação dos personagens em um jogo de RPG.

O primeiro passo que devemos realizar para criar uma função é analisar a estrutura do código que se repete e que justifica o uso de função. Também se devem observar as informações que se alteram entre as repetições, pois são fortes candidatas a serem parâmetros. Por fim, verificar se o resultado do processamento deverá ser retornado. Vamos realizar essas ações no código de movimentação apresentado anteriormente:

```
x_personagem_principal = 10
y_personagem_principal = 10

x_personagem_ferreiro = 5
y_personagem_ferreiro = 20

# Coordenadas da edificação
x_edificacao = 15
y_edificacao = 10

# Movimenta-se o personagem principal em 5 pontos da coordenada x
x_personagem_principal += 5

if x_personagem_principal == x_edificacao and y_personagem_principal  ==
y_edificacao:
    print("Aconteceu uma colisão")

# Movimenta-se o personagem ferreiro em 15 pontos da coordenada x
x_personagem_ferreiro += 15

if x_personagem_ferreiro == x_edificacao and y_personagem_ferreiro  ==
y_edificacao:
    print("Aconteceu uma colisão")
```

Observe que ambos os personagens possuem suas coordenadas x modificadas e, após isso, há uma verificação de colisão. Esses são os códigos que se repetem e que foram mencionados anteriormente. A informação que varia entre os códigos é a coordenada atual do

personagem e valor de movimentação aplicado a ela. Essas informações serão os nossos parâmetros. Por fim, é interessante que após a movimentação a nova coordenada seja retornada para que possa ser utilizada na atualização da coordenada anterior do personagem.

O código a seguir apresenta uma função com os elementos descritos no parágrafo anterior:

```
x_personagem_principal = 10
y_personagem_principal = 10

x_personagem_ferreiro = 5
y_personagem_ferreiro = 20

x_edificacao = 15
y_edificacao = 10

def movimentar( coordenada_x_atual, coordenada_y, deslocamento ):
    nova_posicao = coordenada_x_atual + deslocamento
    if nova_posicao == x_edificacao and coordenada_y  == y_edificacao:
         print("Aconteceu uma colisão")
    return nova_posicao

x_personagem_principal = movimentar( x_personagem_principal,
y_personagem_principal, 5 )
x_personagem_ferreiro = movimentar( x_personagem_ferreiro,
y_personagem_ferreiro, 15 )
```

Criamos uma função nomeada *movimentar* que recebe três parâmetros: a coordenada x atual do personagem, o movimento que será realizado no eixo x e a coordenada y. As ações que se repetiam foram incorporadas ao escopo desta função. Observe como a função foi utilizada para o personagem principal e o ferreiro, com apenas duas linhas de código. Esse é um dos grandes benefícios das funções, pois, após criada, ela pode ser utilizada quantas vezes forem necessárias.

O código apresentado apresenta alguns benefícios em comparação ao do início deste capítulo. Primeiramente, reduzimos as instruções que se repetiam em nosso código, tornando-o mais legível. Em segundo lugar, as operações de movimentação e colisão ficam concentradas em um único local, o que facilita a sua manutenção, pois se as regras de movimentação forem alteradas basta modificar o escopo da função. Por fim, se houver outros personagens que se movimentam podemos utilizar essa mesma função, evitando a escrita de mais código.

Para finalizar, é preciso estar ciente que modificações em uma função já criada e em uso, podem ser perigosas. Vamos ilustrar um dos problemas que podem acontecer, considere o código a seguir da função `dizer_ola` :

```
def dizer_ola( nome_usuario ):
    print("Olá", nome_usuario)

dizer_ola("Pietra")
```

Considere agora uma mudança na assinatura desta função que passa a receber dois parâmetros:

```
def dizer_ola( nome_usuario, sobrenome):
    print("Olá", nome_usuario, sobrenome)

dizer_ola("Pietra")
```

Observe que, apesar da mudança da assinatura da função, esquecemos propositalmente, neste caso, de alterar a sua execução. Consequentemente um erro será ocasionado: `TypeError: dizer_ola() missing 1 required positional argument: 'sobrenome'` . Portanto, mudanças na assinatura da função podem demandar modificações nos trechos de código que realizam a sua execução.

Há ainda outros problemas a serem considerados. Por exemplo, quando criamos a função *movimentar*, ela era utilizada pelo personagem principal e o ferreiro. Agora considere que por alguma

razão o movimento do personagem principal mudou e você resolve alterar o escopo desta função com as modificações. Como consequência, essas mudanças também serão aplicadas ao personagem ferreiro, que utiliza a mesma função para movimentação. Esse impacto indesejado pode comprometer o funcionamento do nosso código. Percebe-se que, como as instruções não são mais compartilhadas, é preciso criar uma outra função de movimentação exclusivamente para o personagem principal.

Pelos motivos apresentados é preciso estar atento às modificações nas funções para evitar "quebras" em seu código. Isso não significa que suas funções não devem mudar, mas é preciso estar ciente dos seus impactos.

Questões

1) Qual a sintaxe para retorno de um valor em uma função?

2) Por que devemos utilizar retorno em nossas funções?

3) Em quais situações devemos utilizar retorno?

Respostas

1) Qual a sintaxe para retorno de um valor em uma função?

R: `return valor`

2) Por que devemos utilizar retorno em nossas funções?

R: O uso de retorno permite que uma função crie uma resposta a partir do seu processamento. Essa resposta poderá ser utilizada no código que realizou a chamada da função.

3) Em quais situações devemos utilizar retorno?

R: Quando a função realiza algum tipo de processamento cujo resultado é de interesse de quem a executou.

9.6 Escopo e tempo de vida das variáveis

O conceito de retorno pode ser um pouco confuso de compreender, pois requer o entendimento sobre dois temas: como o Python executa as instruções de um escopo da função e como as variáveis criadas nele são gerenciadas.

Iniciamos pela explicação sobre o fluxo de execução das instruções de uma função. Até o momento sabemos que a execução do nosso código é realizada linha após linha, de cima para baixo e da esquerda para a direita. No entanto, ao utilizar funções, esse fluxo é um pouco diferente.

Na figura a seguir ilustramos um código que contém seis linhas de programação, incluindo uma função. Observe que existem seis tempos que demonstram a execução do código. A seta demonstra, linha após linha, como esse processo ocorre.

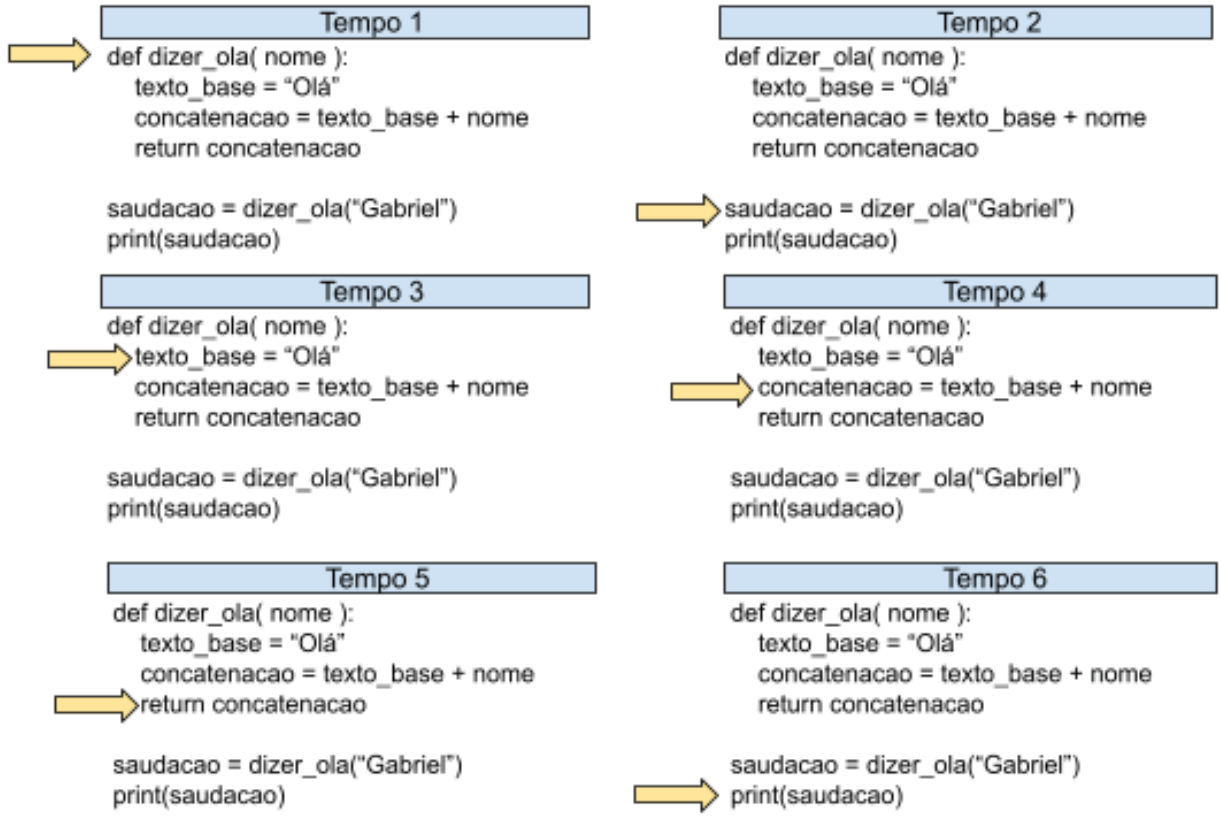

Figura 9.11: Fluxo de execução de um algoritmo com função.

Observe que o fluxo de execução do código não ocorreu de forma completamente linear. Do tempo 1 para o tempo 2, a execução salta das linhas que definem a função para a linha onde ela é executada. Isso ocorre, pois as instruções dentro da função somente são executadas após a função ser chamada (tempo 3). Após isso, o escopo da função é executado linearmente, como você já conhece. Observe, no entanto, que após a instrução de retorno (tempo 5) a execução passa para a última linha do código (tempo 6). Isso faz sentido, pois o retorno marca o término da função e a partir disso a execução segue para a instrução que vem após a chamada da função.

Também é importante compreender como o Python gerencia a memória das variáveis criadas no escopo de uma função.

Abordaremos esse tema de forma simplificada, pois há procedimentos complexos que fogem ao escopo deste livro.

Lembre-se de que o computador guarda nossas variáveis em sua memória RAM. Em algum momento essas variáveis são removidas da memória, seja quando o Python entende que alguma variável em particular não será mais utilizada ou quando seu programa fecha. Em geral não precisamos nos preocupar com isso, pois o Python gerencia isso forma automática e bastante eficaz. No entanto, esse conhecimento é importante para entender como as variáveis são gerenciadas no escopo das funções e a importância do `return` (tema que desencadeou essa discussão).

Na figura a seguir ilustramos dois momentos diferentes da execução da função `dizer_ola`, onde temos o *Tempo n* que representa um momento anterior ao *Tempo n + 1*. A seta indica qual linha do código está em execução naquele momento e as variáveis que estão armazenadas na memória. Observe como essas variáveis são excluídas da memória a depender de onde está a execução do código em um dado momento.

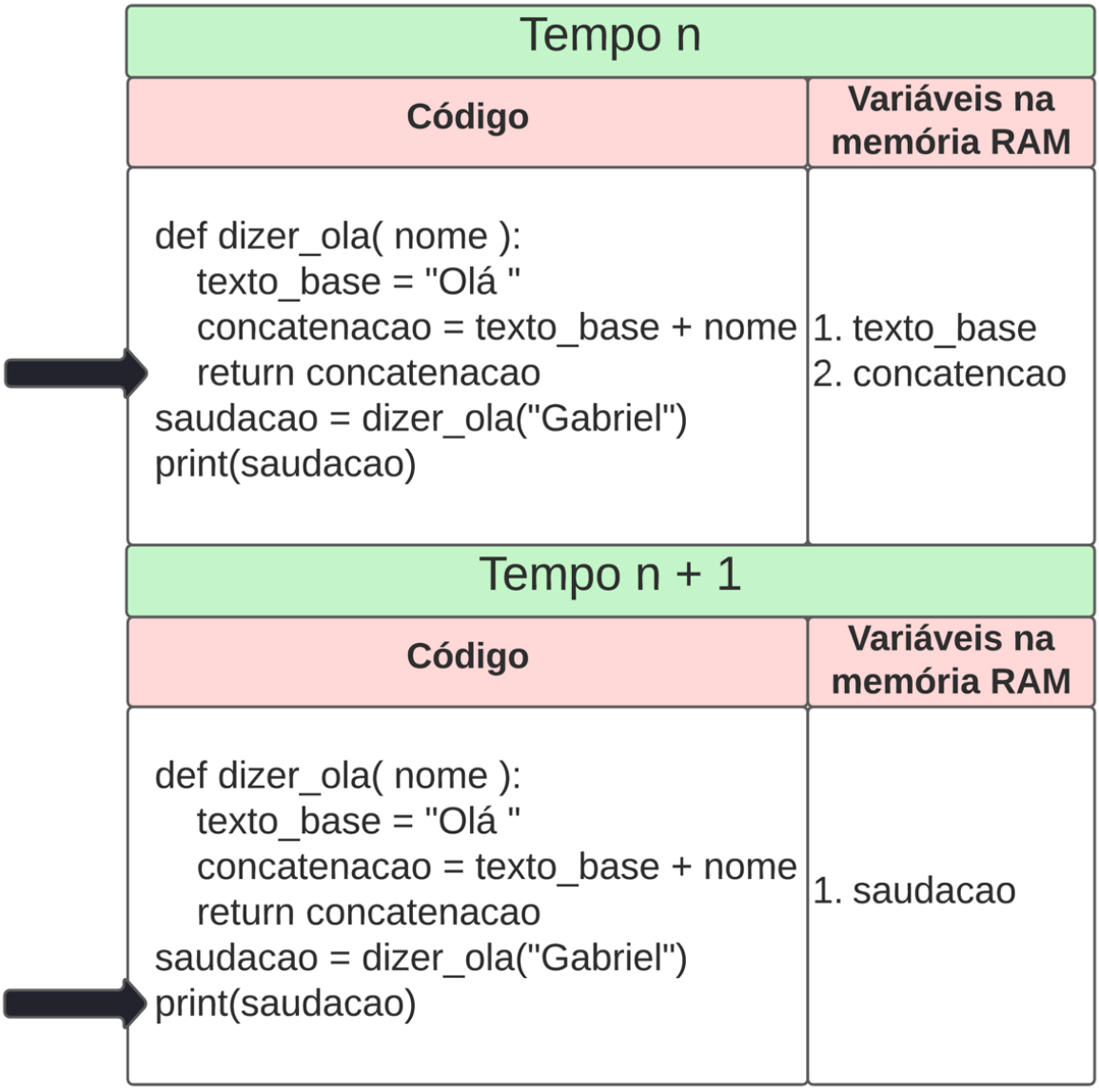

Figura 9.12: Gestão do tempo de vida das variáveis.

No *Tempo n*, a execução do código está na última linha do escopo da função. Neste momento, as duas linhas anteriores já foram executadas, fazendo com que duas variáveis fossem criadas e guardadas na memória RAM do computador. No *Tempo n + 1*, percebe-se que a seta está na última linha do código, sendo assim, a execução da função já foi finalizada e um valor retornado. Temos uma variável `saudacao` que guarda o dado retornado pela função. Observe que neste momento as variáveis do *Tempo n* que estavam

na memória foram excluídas, pois após o retorno da função o Python se encarrega de eliminar tudo o foi criado em seu escopo. Assim, o `return` foi útil para "retirar" um dado de dentro da função (e que seria excluído da memória RAM na sua finalização) para um local onde continuará existindo.

Percebe-se assim que o tempo de vida de nossas variáveis é definida pelo escopo ao qual elas pertencem, sejam de funções, condições, repetições, entre outros. Esse conhecimento é útil para, além da compreensão sobre a importância do `return`, evitar que alguns erros aconteçam. Por exemplo, considere o código a seguir, que realiza a impressão de uma variável.

```
def apresentar_numero():
    numero = 3
    print(numero)

apresentar_numero()
print(numero)
```

No escopo da função foi declarada uma variável *numero*, seguido da sua impressão. Após a execução da função, tentou-se imprimir o valor dessa variável. Como a variável *numero* somente existe no escopo da função, um erro será ocasionado: **NameError: name 'numero' is not defined**.

Variáveis que existem dentro de um escopo específico recebem o nome de **local**.

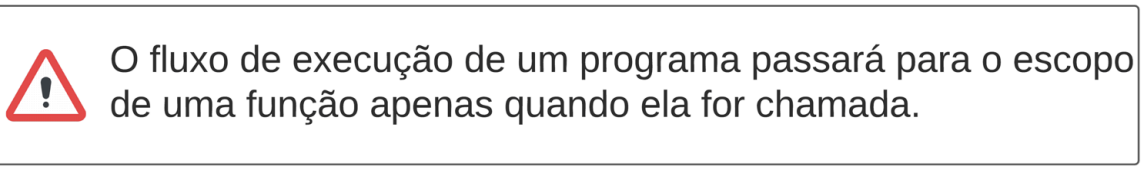

Figura 9.13: O fluxo de execução de um programa passará para o escopo de uma função apenas quando ela for chamada.

Também é importante conhecer um outro tipo de escopo, conhecido por **global**. Observe o código a seguir, que contém uma pequena variação em comparação ao anterior, pois a declaração da variável `numero` inicia-se fora do escopo da função:

```
numero = 3

def apresentar_numero():
    numero = 2
    print(numero)

apresentar_numero()
print(numero)
```

A execução desse código resulta em duas saídas, talvez não esperadas por você: `2` e `3`. Apesar de simples, o código tem uma particularidade em relação ao escopo da variável `numero`. Como dito anteriormente, sua primeira declaração é feita fora do escopo da função (linha 1), o que lhe atribui um **escopo global**. Dentro do escopo da função a variável tem seu valor alterado. No entanto, esta possui um **escopo local**, pois relativo à função. Assim, o Python entende que há duas variáveis diferentes, apesar de compartilharem o mesmo nome, e fará a gestão dos seus tempos de vida de maneira diferente. Por exemplo, a variável criada no escopo da função será excluída após a sua finalização. É por essa razão que o segundo `print` apresenta o valor *3*, pois fez uso do valor da variável `global`.

O Python gerencia o tempo de vida das variáveis a partir do escopo ao qual pertencem.

Figura 9.14: Retorno de uma função.

Em situações em que desejamos explicitamente fazer uso de uma variável global dentro do escopo de função, é preciso utilizar a

instrução **global**. Ela deve ser escrita antes do nome da variável e indicará ao Python que os usos subsequentes da variável fazem referência à variável de escopo global. O código anterior foi modificado para essa finalidade:

```
numero = 3

def apresentar_numero():
    global numero
    print(numero)
    numero = 2
    print(numero)

apresentar_numero()
print(numero)
```

O que produzirá a saída:

```
3
2
2
```

Observe a primeira instrução presente na função. Ela informa que vamos utilizar a variável `numero` declarada no escopo global. Modificações dessa variável dentro da função alterarão o seu valor global.

9.7 Funções Lambda

As funções Lambda são um tipo especial de função nomeadas como anônimas. O entendimento deste tópico requer que os conceitos de funções tenham sido internalizados por você, pois facilitará a comparação com a sintaxe Lambda.

De uma forma resumida, o uso de `lambda` cria uma função em uma única linha e que possui similaridades com as funções tradicionais:

realizam uma ação, recebem parâmetros e produzem um retorno. No entanto, a forma como são criadas difere significativamente de uma função tradicional. Além disso, há casos em particular que justificam o uso do `lambda`, e falaremos sobre isso em breve.

Para conhecer a sintaxe `lambda`, vamos tomar como base um código que realiza a soma de dois números:

```
def somar(numero_um, numero_dois):
    return numero_um + numero_dois

resultado = somar(2,2)
```

O uso de `lambda` permite criar uma função com o mesmo comportamento, mas utilizando uma sintaxe diferente:

```
somar = lambda numero_um, numero_dois:numero_um+numero_dois
resultado = somar(2,2)
```

Perceba como utilizamos apenas uma instrução para representar a função.

A sintaxe para uso de `lambda` é:

```
nome-da-funcao = lambda:parâmetros:expressão
```

Uma limitação no uso de funções `lambda` é que elas **somente podem conter uma instrução**, diferente das funções tradicionais onde não há limite.

Uma pergunta que você pode estar se fazendo é: *quando vou utilizar esse recurso?*. Um dos benefícios das funções `lambda` é que podemos passá-las como parâmetros para outras funções. Isso pode parecer bem complicado de entender, e de fato é, mas vamos explicar.

No código a seguir será criada uma função "tradicional" que realiza a multiplicação de um número por dois. Sabemos que esse tipo de

operação matemática pode ser feito por meio da soma de um número com ele mesmo. Assim, em vez de implementar uma operação de soma dentro da função de multiplicação vamos aproveitar uma função `lambda`:

```
somar = lambda numero_um, numero_dois:numero_um+numero_dois

def multiplicar_por_dois(funcao_somar, numero):
    resultado_multiplicacao = funcao_somar(numero, numero)
    return resultado_multiplicacao

resultado = multiplicar_por_dois(somar, 5)
print(resultado)
```

A saída desse código será `10`, pois o parâmetro informado foi 5. Observe que na linha 1 criamos uma função `lambda` que realiza uma soma entre dois números. Na linha 3 há a assinatura da função `multiplicar_por_dois`, que recebe dois parâmetros. Observe a linha 7 onde a função `multiplicar_por_dois` é chamada. Perceba que o primeiro parâmetro é o nome que demos à função de somar (sem os parênteses). Dessa forma, poderemos fazer uso dessa função dentro do escopo de `multiplicar_por_dois` por meio do parâmetro `funcao_somar`.

É normal que a passagem de funções como parâmetro seja um conceito difícil de compreender e você não deve se assustar. É possível criar programas sem esse recurso, mas é importante conhecê-lo e revisitar esta seção no futuro para melhor assimilar os conceitos apresentados.

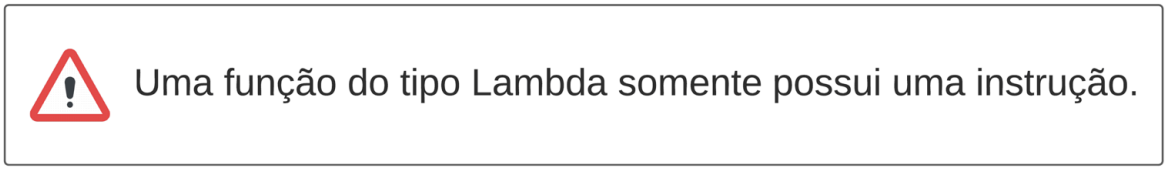

Figura 9.15: Uma função do tipo Lambda somente possui uma instrução.

9.8 Erros mais comuns

Falta de parênteses. Este erro pode ocorrer na declaração ou chamada de uma função:

Código com problema:

```
# declaração de uma função

def somar(numero_um, numero_dois:
    # escopo da função

# chamada de uma função

somar(2,2
```

Código corrigido:

```
# declaração de uma função

def somar(numero_um, numero_dois):
    # escopo da função

# chamada de uma função

somar(2,2)
```

Falta de vírgula para separar parâmetros. Os parâmetros de uma função devem ser separados por vírgulas:

Código com problema:

```
# declaração de uma função
def somar(numero_um numero_dois):
    # escopo da função

# chamada de uma função

somar(2 2)
```

Código corrigido:

```
# declaração de uma função
def somar(numero_um, numero_dois):
    # escopo da função

# chamada de uma função

somar(2,2)
```

Falta de dois pontos. Assim como ocorre com as condições e repetições, o início do escopo de uma função é definido a partir do sinal de : (dois pontos):

Código com problema:

```
# declaração de uma função
def somar(numero_um numero_dois)
    # escopo da função
```

Código corrigido:

```
# declaração de uma função
def somar(numero_um, numero_dois):
    # escopo da função
```

Executar uma função antes de declará-la. Uma função somente pode ser executada após a sua declaração. Assim, essas instruções devem ser escritas antes da chamada.

Código com problema:

```
dizer_ola()

def dizer_ola():
    print("Olá mundo!")
```

Código corrigido:

```
def dizer_ola():
    print("Olá mundo!")
```

```
dizer_ola()
```

Tentar executar uma função informando menos parâmetros. A chamada de uma função deve obedecer ao quantitativo de parâmetros que ela possui.

Código com problema:

```
def dizer_ola(nome):
    print(f"Olá {nome}")

dizer_ola()
```

Código corrigido:

```
def dizer_ola(nome):
    print(f"Olá {nome}")

dizer_ola("Bill Gates")
```

Utilizar uma variável local em um escopo global. Variáveis declaradas dentro de uma função somente existem em seu escopo.

Código com problema:

```
def dizer_ola(nome):
    saudacao = f"Olá {nome}"
    print(saudacao)

dizer_ola("Bill Gates")
print(saudacao)
```

Código corrigido:

```
def dizer_ola(nome):
    saudacao = f"Olá {nome}"
    return saudacao

texto = dizer_ola("Bill Gates")
print(texto)
```

9.9 Resumo e mapa mental

- Neste capítulo aprendemos o que são funções e as situações que justificam o seu uso;
- Uma função executa alguma ação e pode ser utilizada múltiplas vezes (reúso);
- O Python oferece um conjunto de funções prontas para uso;
- Vimos que as funções na programação se dividem entre chamada e criação;
- A chamada de uma função consiste na execução das instruções que pertencem ao seu escopo;
- A sintaxe para criação de uma função é composta por um nome, parâmetros, escopo e retorno.

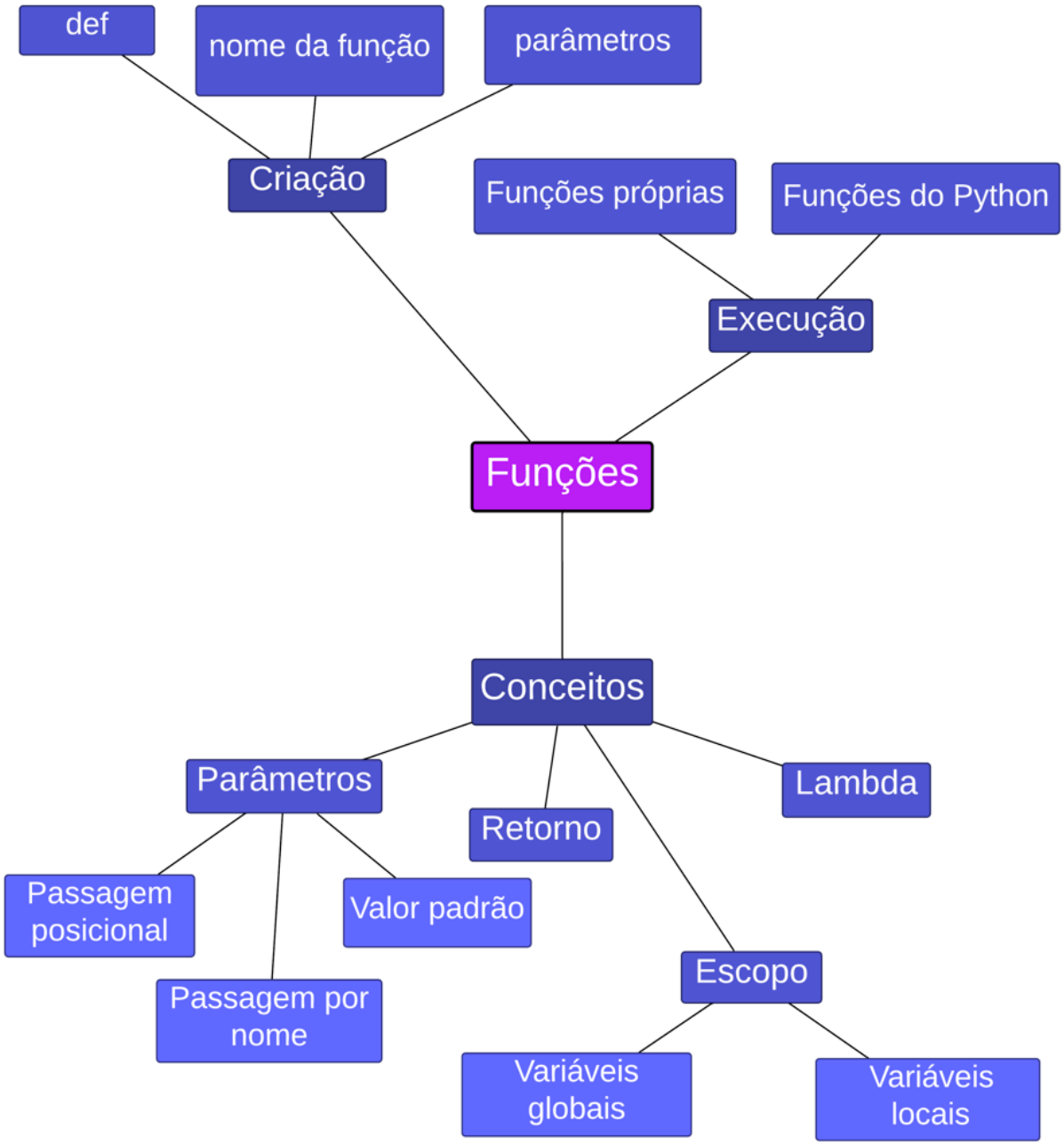

Figura 9.16: Mapa mental do capítulo 9.

Estruturas para organização de dados

Capítulo 10
Listas e Tuplas

1. O que você já sabe?

As linguagens de programação armazenam os dados do programa em variáveis de diferentes tipos: numéricas, *Strings*, booleanas, entre outras. Essas variáveis são úteis, mas apresentam uma grande limitação: somente podem armazenar um dado por vez.

2. O que veremos?

Com frequência os softwares precisam gerenciar dados que possuem alguma relação entre si. Por exemplo, imagine que você precisa armazenar 30 notas dos estudantes de uma turma, para isso seriam necessárias 30 variáveis diferentes. Além de tornar o código extenso, realizar operações nesses dados (como saber a maior nota da turma) pode se tornar uma operação complexa pela necessidade de trabalhar com todas essas variáveis.

Para facilitar, em situações como essa o Python possui um tipo especial de variável denominada **contêiner de dados**. Sua grande vantagem em comparação às variáveis tradicionais é a possibilidade de guardar múltiplos dados em um único local.

3. Por que é importante?

Uma aplicação é formada por dados simples (armazenados nas variáveis que já conhecemos) e também por estruturas mais

complexas. Utilizar variáveis tradicionais para representá-los implica em alguns problemas para a pessoa programadora, que são mitigados com os recursos que serão apresentados neste e no próximo capítulo.

O que você vai aprender?

Ao término deste capítulo você compreenderá:

1. O que são contêineres de dados e em quais situações você deve utilizá-los;
2. Os diferentes tipos de contêineres que o Python oferece e quando cada um deles é aplicado;
3. A sintaxe para criação e uso de contêineres;
4. As operações que podem ser realizadas para manipular conjuntos de dados.

10.1 Qual problema os contêineres de dados resolvem?

Como mencionado no início do capítulo, considere uma situação onde é preciso calcular a média das notas dos alunos de uma turma. Para simplificar, vamos visualizar um exemplo com apenas três notas:

```
nota_um = 10
nota_dois = 5
nota_tres = 7

media  = (nota_um + nota_dois + nota_tres)/3
```

Se pensarmos na representação do mundo real, essas notas possuem uma ligação entre elas, pois pertencem a alunos de uma mesma turma. Além disso, o cálculo da média de uma turma

depende dessas notas, evidenciando a relação que elas possuem entre si.

Voltando ao código, ele não apresenta nenhum problema e está funcional. No entanto, considere que serão calculadas as médias de dez alunos:

```
nota_um = 10
nota_dois = 5.5
nota_tres = 7
nota_quatro = 8
nota_cinco = 6
nota_seis = 7
nota_sete = 9.5
nota_oito = 6.3
nota_nove = 6
nota_dez = 2.4

somatorio_notas = nota_um + nota_dois + nota_tres + nota_quatro +
nota_cinco + nota_seis + nota_sete + nota_oito + nota_nove + nota_dez

quantidade_notas = 10

media  = somatorio_notas/quantidade_notas
```

Claramente o código ficou mais extenso, não é? E se fosse preciso aumentar para 30 notas? É possível imaginar que nosso algoritmo ficaria ainda maior e o cálculo da média se tornaria mais complexo. Situações como essa, em que os dados possuem alguma relação entre eles, são comuns na programação. Por exemplo, os produtos no carrinho de compras de um cliente, todas as mensagens enviadas para um amigo em uma rede social, sua playlist favorita em um aplicativo de streaming, entre outros. Nesses casos, esses dados são melhor gerenciados na programação por meio de uma estrutura conhecida por **contêiner**.

Na prática, um contêiner é um tipo "especial" de variável que permite armazenar um conjunto de dados em um único local. Além

dessa capacidade, eles possuem funcionalidades que facilitam a realização de operações nesses dados, como cálculos matemáticos.

Vamos apresentar este recurso, pois você deve estar curioso para conhecê-lo na prática. Neste momento não se preocupe em entender o código a seguir por completo, o objetivo é apenas ser o primeiro contato com um contêiner. O código a seguir realiza a mesma operação do anterior e calcula a média de uma turma com dez notas, mas utiliza um tipo específico de contêiner conhecido por **lista**.

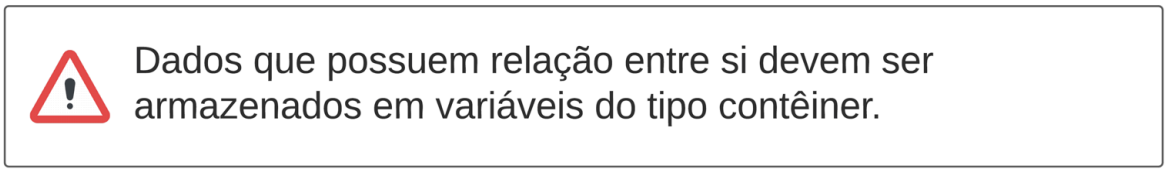

Figura 10.1: Dados que possuem relação entre si devem ser armazenados em variáveis do tipo contêiner.

```
notas = [ 10, 5.5, 7, 8, 6, 7, 9.5, 6.3, 6, 2.4 ]
somatorio_notas = 0
quantidade_notas = len( notas )

for nota in notas:
    somatorio_notas += nota

media = somatorio_notas / quantidade_notas
```

Percebe-se que o código está com menos linhas. Observe também como a sintaxe da variável `notas` difere de uma variável tradicional, pois estamos guardando dez dados diferentes (cada um deve ser separado por vírgula) em um único local. Outra mudança no código é a presença da estrutura de repetição utilizada para iterar pelas notas. Esse procedimento possibilita o acesso individual a cada nota armazenada no contêiner. Nesse caso em especial realizamos isso para somar todas as notas da turma.

Um grande benefício deste código em comparação ao anterior é que, para trabalhar com mais notas, basta incluí-las na variável *notas*, pois o restante do algoritmo está pronto para lidar com essa mudança. Realizar esse mesmo procedimento no código anterior iria demandar a criação de novas variáveis, como também alterar a fórmula de cálculo da média. Por fim, ter apenas uma variável para guardar todas as notas faz muito sentido, em especial quando o programa cresce e passa a ter centenas ou mesmo milhares delas. Isso torna nosso código mais organizado, facilitando a localização dessas informações.

Os contêineres permitem armazenar mais de um dado em uma única variável. Para explicar esse conceito faremos uma analogia com um guarda-roupa cheio de roupas. Esse objeto equivale ao nosso contêiner no Python, pois pode guardar roupas (o equivalente a um dado) que possuem alguma relação entre si (pertencem a você, por exemplo).

Figura 10.2: Um contêiner armazena nossos dados assim como um guarda-roupa pode armazenar diferentes roupas. Crédito: Artem Beliaikin.

Os contêineres violam uma das regras que falamos no capítulo 3: *"Uma variável armazena apenas uma informação."*. Naquele momento era importante entender dessa forma, pois iniciantes na programação apresentam algum tipo de dificuldade com essa interpretação. Ele ainda se aplica, mas apenas às variáveis de tipo numérico, *String* e booleano.

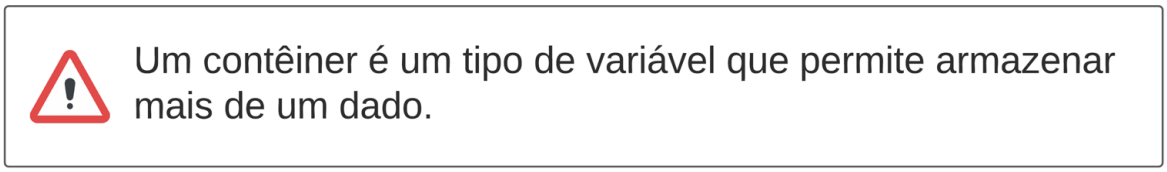

Figura 10.3: Um contêiner é um tipo de variável que permite armazenar mais de um dado.

10.2 Tipos de contêineres padrão do Python

Neste capítulo e no próximo abordaremos os quatro principais contêineres disponibilizados pelo Python: listas, tuplas, conjuntos e dicionários. Cada um deles possui características particulares que os tornam mais adequados para determinadas situações. Vamos conhecê-los a seguir e para facilitar o seu entendimento apresentaremos uma representação visual que ilustra como os dados são organizados em cada contêiner.

Listas. Os dados são organizados de forma linear e cada um recebe uma posição que é utilizada para localizá-lo. Isso permite saber quem é o primeiro ou último dado, como também em qual posição encontra-se um dado em específico. Após criada, uma lista permite a inclusão ou remoção de dados.

Figura 10.4: Representação de uma lista.

Tuplas. São similares às listas com a exceção que após a sua criação, não é permitido adicionar ou excluir dados.

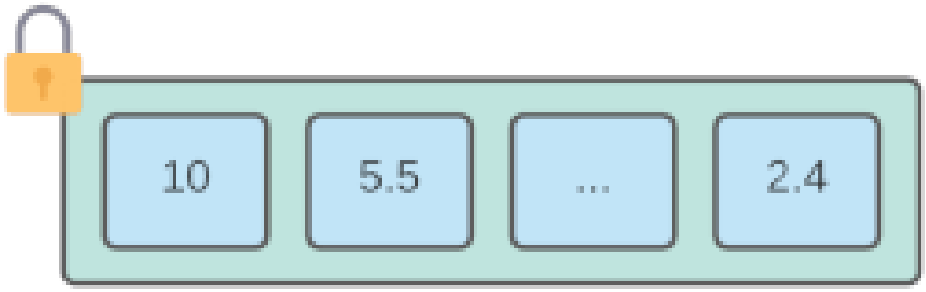

Figura 10.5: Representação de uma tupla.

Dicionário. Os dados guardados em um dicionário são associados a chaves. Pelo gráfico observa-se que a nota 5 (dado) está associada ao estudante Leonardo Soares (chave). Diferente das listas e tuplas onde os dados são localizados por sua posição, esse contêiner possui um mecanismo mais complexo para identificação do dado, que pode ser útil em algumas situações.

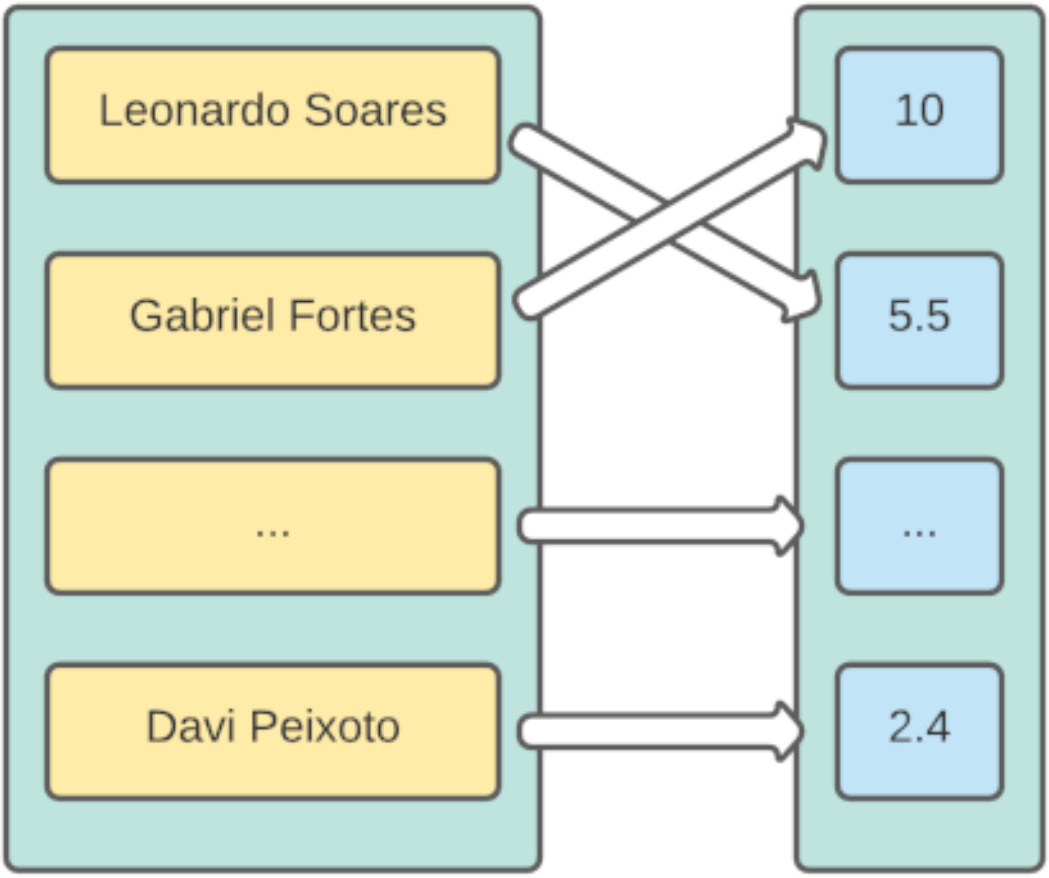

Figura 10.6: Representação de um dicionário.

Conjuntos. Neste contêiner os dados são guardados em uma representação similar a um conjunto matemático. Seu maior benefício é possibilitar a realização de operações entre conjuntos, como intersecções, uniões, entre outros.

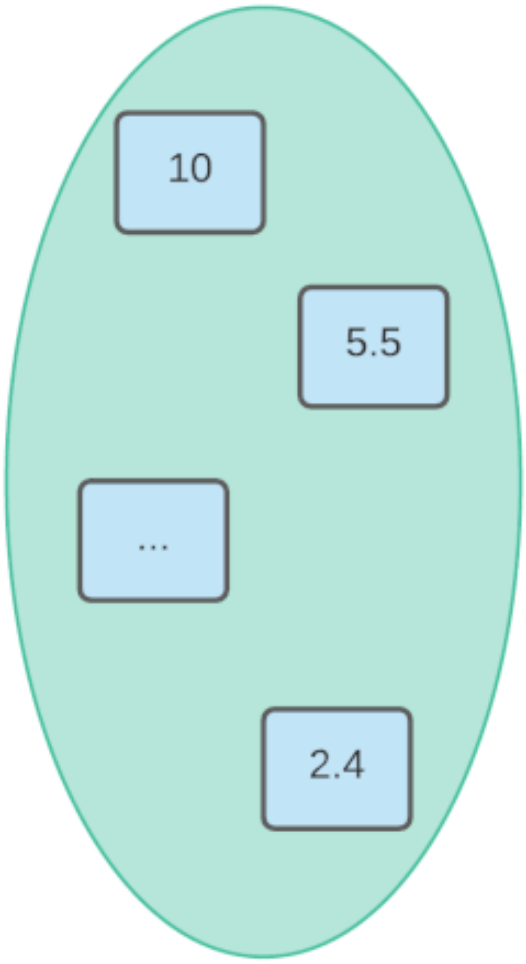

Figura 10.7: Representação de um conjunto.

Alguns conceitos se aplicam a todos os contêineres e são importantes para sua plena compreensão:

- Um contêiner guarda um ou mais dados que são conhecidos por *elementos*;
- Um contêiner é uma variável;
- É possível acessar os elementos de um contêiner individualmente;
- Os contêineres possuem um tamanho que é igual ao total de elementos que ele possui.

Lembre desses conceitos, pois eles serão aplicados nas seções seguintes.

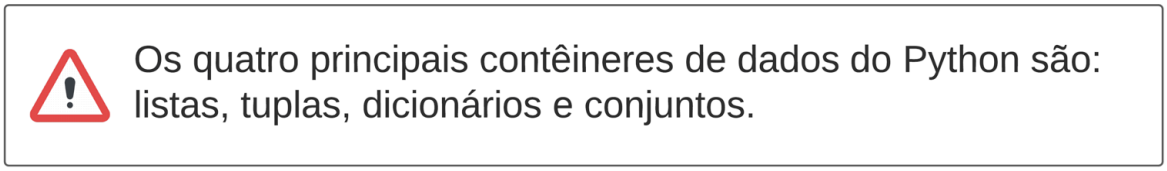

Figura 10.8: Os quatro principais contêineres de dados do Python são: listas, tuplas, dicionários e conjuntos.

Questões

1) Quais os principais contêineres de dados do Python?

2) O que são contêineres de dados no Python?

3) Qual problema os contêineres resolvem?

4) Quais as características dos diferentes tipos de contêineres existentes no Python?

Respostas

1) Quais os principais contêineres de dados do Python?

R: Lista, tupla, dicionário e conjuntos.

2) O que são contêineres de dados no Python?

R: É um tipo de variável que permite armazenar mais de um dado.

3) Qual problema os contêineres resolvem?

R: Quando há vários dados que possuem alguma relação entre si, como os itens de uma feira de supermercado, armazená-los em diferentes variáveis torna o nosso código extenso e desorganizado.

Além disso, os contêineres possuem funcionalidades que permitem a manipulação de uma coleção de dados mais facilmente.

4) Quais as características dos diferentes tipos de contêineres existentes no Python?

R: Listas e Tuplas armazenam um conjunto de dados. No entanto, as tuplas não podem ser alteradas após criadas. Dicionários associam cada dado a uma chave, o que confere uma melhor organização à informação, quando isso é desejado. Por fim, os conjuntos guardam os dados de forma não sequencial e são úteis em situações em que se deseja realizar operações entre diferentes conjuntos, como intersecção, união, entre outras.

10.3 Listas

Como o nome sugere, esse contêiner é utilizado para armazenar uma lista de dados. Destacamos alguns exemplos onde esse recurso poderia ser aplicado: criar uma lista de compras; armazenar as notas de alunos em uma turma, onde não interessa saber de quem é cada nota; guardar os nomes dos clientes de uma academia.

Há duas sintaxes para declarar uma lista e ambas utilizam um colchete de abertura e fechamento:

```
nome-da-lista = [ ]
```

Essa sintaxe é utilizada para declarar uma lista vazia, sem elementos. É útil para situações em que não sabemos previamente os dados que farão parte da lista. Por exemplo, você criou um aplicativo para calcular a média das notas de uma turma, mas quem

fornecerá os dados é o professor (usuário) quando executar sua aplicação.

Também é possível declarar uma lista e iniciá-la com elementos. Neste caso, os dados são informados dentro do colchetes e separados por vírgulas, como no exemplo anterior das notas dos alunos de uma turma:

```
notas = [ 10, 5.5, 7, 8, 6, 7, 9.5, 6.3, 6, 2.4 ]
```

Uma lista pode armazenar qualquer tipo de dado existente no Python. Também é possível mesclar diferentes tipos em uma mesma lista:

```
lista = ["a", 2, True]
```

Acesso e modificação dos elementos

Cada elemento em uma lista possui uma posição que indica a sua localização. Essa posição também é chamada de índice, inicia-se em zero (primeiro elemento da lista) e incrementa de um em um. A figura a seguir ilustra esse conceito por meio de uma lista com 10 elementos. Por enquanto você deve ignorar o índice negativo, pois falaremos dele em breve.

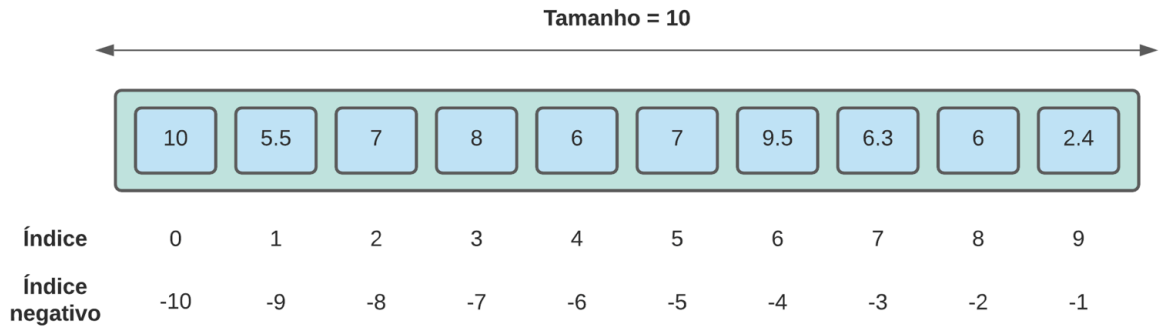

Figura 10.9: Partes de uma lista.

O índice é utilizado para acessar um elemento específico na lista. Para isso, utiliza-se a seguinte sintaxe:

```
nome-da-lista[ índice ]
```

O elemento recuperado pode ser utilizado diretamente ou armazenado em uma variável. O código a seguir representa o acesso ao terceiro elemento de uma lista.

```
notas = [ 10, 5.5, 7, 8, 6, 7, 9.5, 6.3, 6, 2.4 ]
print( notas[2] )
```

Isso resultará na saída: `7` . Observe que a sintaxe `notas[2]` recuperou o terceiro elemento da lista. Os índices confundem muitos aprendizes que acreditavam que seria recuperado o segundo elemento. É preciso lembrar que o índice de uma lista inicia-se em 0. Portanto, a posição 2 representa o terceiro elemento (0, 1, 2).

Também é possível guardar um elemento em uma variável por meio da sintaxe:

```
variavel = nome-da-lista[ índice ]
```

Como exemplificado a seguir:

```
notas = [ 10, 5.5, 7, 8, 6, 7, 9.5, 6.3, 6, 2.4 ]
nota_terceiro_aluno = notas[2]
```

O primeiro índice de um elemento é 0 e o último é n – 1.

Figura 10.10: O primeiro índice de um elemento é 0 e o último é n – 1.

Para modificar um elemento da lista é preciso saber o seu índice e seguir a sintaxe:

```
nome-da-lista[ índice ] = novo-valor
```

No código a seguir alteramos o primeiro elemento de uma lista:

```
notas = [ 10, 5.5, 7, 8, 6, 7, 9.5, 6.3, 6, 2.4 ]
notas[0] = 8
print(notas)
```

Isso produzirá a saída: `[ 8, 5.5, 7, 8, 6, 7, 9.5, 6.3, 6, 2.4 ]`

É importante saber que a variável que guarda um elemento recuperado da lista **não** possui relação com ele. Portanto, a alteração de um não impactará no outro, como pode ser visto no código a seguir. Nele, recuperamos o terceiro elemento (linha 2), modificamos o elemento original (linha 3) e na sequência imprimimos os seus valores para comprovar que são diferentes.

```
notas = [ 10, 5.5, 7, 8, 6, 7, 9.5, 6.3, 6, 2.4 ]
nota_terceiro_aluno = notas[2]
notas[2] = 8
print(nota_terceiro_aluno)
print(notas[2])
```

O código apresentará a saída:

```
7
8
```

É possível acessar os elementos de uma lista de trás para frente utilizando **índices negativos**. Assim, o último elemento é acessado pelo índice -1; o penúltimo elemento, pelo índice -2, e assim sucessivamente:

```
notas = [ 10, 5.5, 7, 8, 6, 7, 9.5, 6.3, 6, 2.4 ]
ultima_nota = notas[-1]
```

Lembra da figura onde apresentamos os índices de uma lista, incluindo os negativos? Sugerimos visualizá-la novamente para compreender as diferenças entre eles.

Por fim, o Python lançará um erro do tipo `IndexError` ao tentar acessar um elemento em um índice inexistente:

```
notas = [ 10, 5.5, 7, 8, 6, 7, 9.5, 6.3, 6, 2.4 ]
print( notas[10] ) # Posição inexistente, pois o último elemento está no índice 9.
```

O código apresentará o erro: `IndexError: list index out of range` .

Acesso a um subgrupo de elementos

Para além do acesso individual aos elementos, é possível recuperar um subgrupo de elementos em uma lista. Por exemplo, podemos acessar as 3 primeiras notas da lista e colocá-las em outra lista. Em casos como esse, utiliza-se o operador `slice` (corte), que **permite recuperar os elementos dentro de um intervalo de índices**. Vamos explicar a sua sintaxe a partir do seu uso:

```
notas = [ 10, 5.5, 7, 8, 6, 7, 9.5, 6.3, 6, 2.4 ]
tres_primeiras_notas = notas[0:3]
```

Observe a segunda linha, onde a variável `tres_primeiras_notas` está sendo declarada. Seu valor é o resultado da operação de `slice` que recupera os três primeiros elementos da lista *notas*, resultando em uma lista de valores `[10, 5.5, 7]` . **Atenção:** a ação de `slice` não exclui os elementos da lista original.

A sintaxe da operação de `slice` é:

> nova-lista-com-subgrupo = nome-nome-da-lista[posicao-inicial : posicao-final]

A operação de corte vai recuperar os elementos compreendidos entre a `posicao-inicial` até o elemento em `posicao-final` subtraído

1. Por exemplo, a sintaxe `notas[0:3]` vai recuperar os elementos de índices 0, 1 e 2. Os dados são guardados na variável `nova-lista-com-subgrupo`.

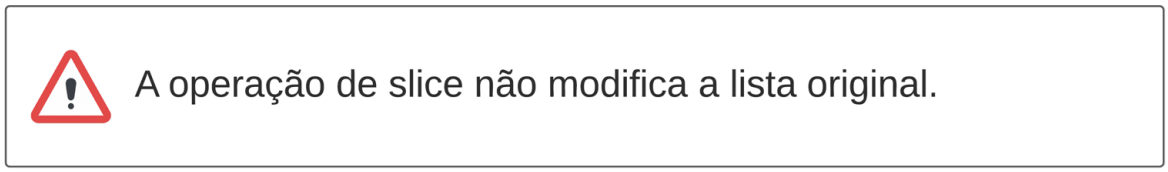

Figura 10.11: A operação de slice não modifica a lista original.

Há uma sintaxe especial do `slice` que pode ser utilizada para realizar a cópia de uma lista. Veja no exemplo a seguir:

```
lista = [1,2,3]
nova_lista = lista[:]
```

Com a sintaxe `[:]`, o Python vai obter todos os elementos de uma lista e criar uma cópia deles para guardá-los em uma outra variável, neste exemplo nomeada de `nova_lista`. Vale lembrar que as duas listas não possuem relação entre si, ou seja, a alteração em uma não impactará na outra.

Inserção de elementos

Uma característica das listas é que elas são **mutáveis**, ou seja, após a sua criação é possível adicionar ou remover elementos. O Python nos oferece um conjunto de funções que facilitam esse processo. A primeira delas é o `append`, que permite incluir um elemento no final da lista. Utiliza-se a seguinte sintaxe:

```
nome-da-lista.append( valor )
```

O código a seguir adiciona o elemento 3 à lista armazenada na variável `notas`.

```
notas = [ 10, 5.5, 7, 8, 6, 7, 9.5, 6.3, 6, 2.4 ]
notas.append(3)
print(notas)
```

O código apresenta a seguinte saída: `[ 10, 5.5, 7, 8, 6, 7, 9.5, 6.3, 6, 2.4, 3 ]`.

Para adicionar um elemento em uma posição específica, podemos utilizar a função `insert`. Ela recebe dois parâmetros: o primeiro é um número inteiro que representa a posição onde o elemento será inserido e o segundo parâmetro será o seu valor.

nome-da-lista.insert(índice, valor)

O código a seguir adiciona o elemento 8 na posição 1 da lista armazenada na variável `notas`.

```
notas = [ 10, 5.5, 7, 8, 6, 7, 9.5, 6.3, 6, 2.4 ]
notas.insert(1, 8)
print(notas)
```

Resultará na saída `[ 10, 8, 5.5, 7, 8, 6, 7, 9.5, 6.3, 6, 2.4 ]`. Observe que o Python precisou deslocar os elementos após o índice informado. Anteriormente o elemento 5.5 estava na posição 1 e passou para a posição 2, e assim sucessivamente.

Remoção de elementos

Há quatro instruções para remoção de elementos em uma lista: `remove`, `pop`, `del` e `clear`. A seguir, apresenta-se cada uma delas.

A função `remove` recebe como parâmetro um valor que será excluído da lista. Caso existam dois ou mais elementos com esse mesmo valor, apenas a primeira ocorrência será removida. Sua sintaxe é representada da seguinte forma:

```
nome-da-lista.remove( valor )
```

Veja um exemplo:

```
notas = [ 10, 5.5, 7, 8, 6, 7, 9.5, 6.3, 6, 2.4 ]
notas.remove(7)
print(notas)
```

Observe que na lista inicial há dois números 7, mas apenas o primeiro é removido, resultando na saída: `[ 10, 5.5, 8, 6, 7, 9.5, 6.3, 6, 2.4 ]`.

Diferente da função `remove`, que exclui a partir de um valor, o `pop` vai realizar a remoção de um elemento pelo seu índice. A função apresenta a sintaxe:

```
nome-da-variavel = nome-da-lista.pop( parâmetro-opcional )
```

A função pode ser utilizada de duas formas diferentes, com ou sem parâmetros. No primeiro formato, deve-se informar um número inteiro que representa o índice do elemento que se deseja remover. Por outro lado, se nenhum parâmetro for informado, o último elemento da lista será removido. A função `pop` vai retornar o elemento excluído (que pode ser útil, caso queira fazer algo com ele) ou lançar um erro `IndexError` caso não exista um elemento associado à posição informada.

No código a seguir será excluído um elemento de índice 1, ou seja, o segundo da lista. Observe como guardamos em uma variável o elemento que foi removido.

```
notas = [ 10, 5.5, 7, 8, 6, 7, 9.5, 6.3, 6, 2.4 ]
elemento_excluido = notas.pop(1)
print(notas)
print(elemento_excluido)
```

A execução do código resulta na saída:

```
[10, 7, 8, 6, 7, 9.5, 6.3, 6, 2.4]
5.5
```

O uso de `pop` sem argumentos vai excluir o último elemento da lista.

```
notas = [ 10, 5.5, 7, 8, 6, 7, 9.5, 6.3, 6, 2.4 ]
notas.pop()
print(notas)
```

Apresenta a saída: `[10, 5.5, 7, 8, 6, 7, 9.5, 6.3, 6]`.

Por fim, ao tentar excluir um elemento em um índice inválido um erro do tipo `IndexError` é ocasionado:

```
notas = [ 10, 5.5, 7, 8, 6, 7, 9.5, 6.3, 6, 2.4 ]
notas.pop(11)
```

O código ao ser executado apresentará a seguinte mensagem de erro: `IndexError: pop index out of range`.

Outra forma de excluir elementos em uma lista é por meio da instrução `del`. Ela pode ser utilizada para excluir elementos em um índice específico, similar ao `pop`. Entretanto, o `del` não retorna o elemento excluído e sua sintaxe é diferente:

del nome-da-lista[índice]

É preciso informar o índice do elemento que será excluído. Veja um exemplo:

```
notas = [ 10, 5.5, 7, 8, 6, 7, 9.5, 6.3, 6, 2.4 ]
del notas[0]
print(notas)
```

O resultado desse código será: `[ 5.5, 7, 8, 6, 7, 9.5, 6.3, 6, 2.4 ]`

A instrução `del` também pode ser utilizada com `slice` e realizar a remoção de um subgrupo de elementos:

```
notas = [ 10, 5.5, 7, 8, 6, 7, 9.5, 6.3, 6, 2.4 ]
del notas[1:6]
print(notas)
```

Esse código vai excluir os elementos de índice 1 até 5 (lembrando que o último número do `slice` é subtraído de um) e resultará na lista: `[ 10, 9.5, 6.3, 6, 2.4 ]`

Também possível excluir todos os elementos de uma lista por meio da seguinte sintaxe:

```
notas = [ 10, 5.5, 7, 8, 6, 7, 9.5, 6.3, 6, 2.4 ]
del notas[:]
print(notas)
```

Com esse código, a variável `notas` perderá todos os seus elementos e se tornará uma lista vazia. Esse mesmo resultado também pode ser alcançado por meio da função `clear` :

```
notas = [ 10, 5.5, 7, 8, 6, 7, 9.5, 6.3, 6, 2.4 ]
notas.clear()
print(notas)
```

A saída após execução será uma lista vazia: `[ ]` .

Iteração pelos elementos

É comum querer acessar não apenas um, mas todos os elementos de uma lista. No início do capítulo apresentamos um caso para calcular a média dos alunos da turma, o que demandou utilizar todas as notas guardadas na lista.

Situações como a apresentada envolvem a iteração pela coleção de dados. Esse procedimento é realizado por meio da instrução `for` e seguindo a sintaxe:

for elemento in nota-da-lista:

Exemplifica-se seu uso no código a seguir:

```
notas = [ 10, 5.5, 7, 8, 6, 7, 9.5, 6.3, 6, 2.4 ]
somatorio_notas = 0
quantidade_notas = len( notas )

for nota in notas:
    somatorio_notas += nota

media = somatorio_notas / quantidade_notas
```

Na primeira linha do código, há a criação da lista de notas que será a base para o cálculo da média. Observe a linha 3, pois há uma função que ainda não foi apresentada: `len`. Ela calcula o total de elementos em uma lista informada em seu parâmetro. Esse dado é necessário para o cálculo da média, pois é o seu divisor.

Observe a sintaxe da repetição criada com o `for` e como ela difere dos exemplos apresentados nos capítulos anteriores. Você pode estar sentindo falta do `range` para definir a quantidade de repetições e, de fato, não precisamos dele. Uma das características dos contêineres é que eles podem ser percorridos diretamente pelo `for`. Pode parecer confuso e explicaremos como a instrução `for` aqui poderia ser lida em português para auxiliar no entendimento:

PARA CADA nota EM notas REPITA:

Assim, o que o Python fará é percorrer cada elemento da lista, iniciando pelo índice zero até o último. Esse processo ocorre por iterações, ou seja, na primeira iteração a variável `notas` guardará o valor 10 (primeiro elemento da lista), na segunda iteração seu valor passa a ser 5.5, e assim sucessivamente até o último elemento.

Também seria possível iterar pela lista utilizando o `while` ainda que isso não seja comum pelo maior trabalho:

```
notas = [ 10, 5.5, 7, 8, 6, 7, 9.5, 6.3, 6, 2.4 ]
somatorio_notas = 0
contador = 0
quantidade_notas = len( notas )

while contador < quantidade_notas:
    somatorio_notas += notas[contador]
    contador += 1

media = somatorio_notas / quantidade_notas
```

Observe como o código é mais extenso, pois precisamos realizar manualmente operações que o `for` realizou automaticamente, como controlar a quantidade de iterações, armazenamento de cada elemento da lista e encerramento da repetição.

Compreensão de listas

Por vezes precisamos inicializar uma lista com uma grande quantidade de elementos, como uma coleção de cem números, inicializados em 0 até o 99: [0, 1, 2 … 99]. Poderíamos fazer de diferentes formas, escrevendo um a um:

```
nova_lista = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34,
35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52,
53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70,
71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88,
89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]
```

Como também poderíamos utilizar o `for` e `range` para geração destes números e inclusão na lista:

```
nova_lista = []
for numero in range(100):
    nova_lista.append(numero)
```

Ainda que os códigos apresentados sejam funcionais, há formas mais eficientes de alcançar o mesmo resultado. A ideia é simplificar o processo de criação e inicialização de listas por meio do conceito de **compreensão de listas** ou, em inglês, *list comprehension*.

Observe a aplicação deste recurso para inicialização da lista com 100 números (explicaremos sua sintaxe em breve):

```
nova_lista = [ numero for numero in range(100) ]
```

Percebe-se como o código é bastante simples e utiliza apenas uma linha de código.

A sintaxe para *list comprehension* é representada da seguinte forma:

nome-da-lista = [expressão for item in iterador]

A variável `nome-da-lista` vai guardar a lista que foi criada. Observe os colchetes indicando a criação da lista e, dentro dele, a instrução utilizada para inicialização dos seus valores. Compreender isso pode ser um pouco complexo e vamos explicar em detalhes a partir da imagem a seguir.

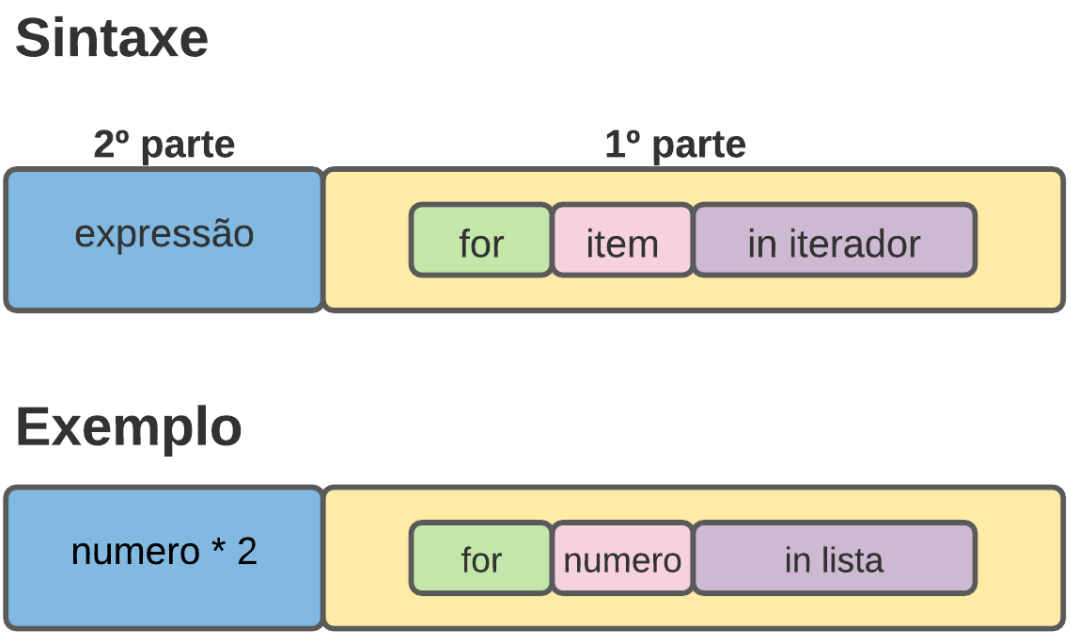

Figura 10.12: Conjuntos com os funcionários dos escritórios.

Dividimos a instrução em duas partes para facilitar a sua compreensão. Elas devem ser lidas de trás para frente, iniciando pelo entendimento do `for` e suas partes, pois é a partir disso que os dados serão gerados. Observe que um iterador é utilizado para fornecer a sequência de dados que darão origem aos elementos da lista. Ele pode ser uma outra lista (como no exemplo), a função `range`, uma tupla, entre tipos de dados que possam ser iterados. Em cada iteração do `for`, um dos dados do iterador será armazenado em uma variável (nomeada `item`), que fica disponível para ser utilizada em `expressão`. No exemplo, nós multiplicamos esse dado por dois e o resultado é adicionado à nova lista, com esse procedimento se repetindo até o fim da iteração.

Compreensão de listas representa uma nova forma para criação e inicialização dos elementos de uma lista.

Figura 10.13: Compreensão de listas representa uma nova forma para criação e inicialização dos elementos de uma lista.

Compreensão de listas também pode ser aplicada para geração de elementos a partir de cálculos matemáticos nos elementos de outra lista. Calma, vamos explicar!

```
lista = [ 0, 1, 2, 3, 4, 5 ]
nova_lista = [ numero * 2 for numero in lista ]
print(nova_lista)
```

Observe como os elementos da variável `lista` são utilizados para criação de `nova_lista`. Esse processo é feito por meio de uma multiplicação por dois aplicada à cada elemento da primeira lista. O resultado é uma lista: `[0, 2, 4, 6, 8, 10]`.

Para finalizar este tema, a compreensão de listas pode contemplar instruções condicionais `if` e `else`, possibilitando gerar elementos para a lista a partir de algum filtro aplicado aos dados do iterador. Para o uso do `if` utiliza-se a seguinte sintaxe:

```
nome-da-lista = [ expressão for item in iterador if comparação ]
```

A aplicação dessa sintaxe é representada a seguir, com criação de uma lista formada apenas por números pares. Vale lembrar que um número é par quando a sua divisão por dois resulta em zero, o que pode ser avaliado com a instrução: `if numero % 2 == 0`.

Assim, o código final é:

```
nova_lista = [ numero for numero in range(100) if numero % 2 == 0]
print(nova_lista)
```

Sua execução resulta em uma lista de elementos: `[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98]`.

Por fim, é possível incluir a instrução `else` para indicar qual operação deve ser realizada quando a comparação do `if` não resultar em verdadeiro. O código a seguir é uma modificação do anterior e forma uma lista com números pares e substitui os ímpares por por -1:

```
nova_lista = [ numero if numero % 2 == 0 else -1 for numero in range(100)
]
print(nova_lista)
```

Isso produz uma lista com os valores `[0, -1, 2, -1, 4, -1, 6, -1, 8, -1, 10, -1, 12, -1, 14, -1, 16, -1, 18, -1, 20, -1, 22, -1, 24, -1, 26, -1, 28, -1, 30, -1, 32, -1, 34, -1, 36, -1, 38, -1, 40, -1, 42, -1, 44, -1, 46, -1, 48, -1, 50, -1, 52, -1, 54, -1, 56, -1, 58, -1, 60, -1, 62, -1, 64, -1, 66, -1, 68, -1, 70, -1, 72, -1, 74, -1, 76, -1, 78, -1, 80, -1, 82, -1, 84, -1, 86, -1, 88, -1, 90, -1, 92, -1, 94, -1, 96, -1, 98, -1]`.

Observe que a sintaxe é diferente da anterior, como pode ser observado a seguir:

> nome-da-lista = [expressão if comparação else expressão for item in iterador]

Operações em listas

Como falamos anteriormente, um dos benefícios dos contêineres é a manipulação de um conjunto de dados de forma integrada. Vimos

isto quando utilizamos o `for` para iterar sobre todos os elementos de uma lista.

Nesta seção destacamos outras funcionalidades que são úteis para as listas.

- **sort**

Utilizado para ordenar os elementos de uma lista, caso exista uma ordem. Ao executá-la, a posição de cada elemento na lista original é alterada para refletir a ordenação.

```
z = [2, 1, 4, 3, 4]
z.sort()
print(z)
```

Isso resultará na saída: `[1, 2, 3, 4, 4]`

- **+**

O operador `+` é utilizado para concatenar uma ou mais listas.

```
z = [2, 1, 4, 3, 4]
y = [5, 6]
z = z + y
print(z)
```

A lista *z* será uma formada pelos elementos: `[2, 1, 4, 3, 4, 5, 6]`

- **in**

Operador para verificar se um elemento está presente na lista:

```
z = [2, 1, 4, 3, 4]
if 4 in z:
    print("O número 4 está presente na lista")
```

Resultará na saída: `O número 4 está presente na lista .`

- **count**

Contabiliza a quantidade de ocorrências de um determinado elemento passado como parâmetro.

```
z = [2, 1, 4, 3, 4]
contagem = z.count(4)
print( contagem )
```

Produzirá como saída: `2`

- **reverse**

Inverte a ordem dos elementos de uma lista.

```
z = [1, 4, 6, 7, 9]
z.reverse()
print( z )
```

Produzirá como saída: `[9, 7, 6, 4, 1]`.

- **index**

Retorna o índice de um elemento em específico.

```
animais = ["cachorro", "gato", "macaco", "gavião"]
indice_gato = animais.index("gato")
print( indice_gato )
```

Produzirá como saída: `1` .

Apresentamos apenas algumas das muitas funcionalidades existentes. Uma relação completa pode ser consultada na documentação oficial (http://docs.python.org).

Questões

1) Qual a sintaxe para criação de uma lista com e sem elementos?

2) Os conceitos de elemento, tamanho da lista e índice fazem parte das listas. O que cada um deles significa?

3) Crie uma lista formada somente por vogais.

4) Crie uma lista formada por números ímpares de 0 a 100. Para isso utilize o conceito de compreensão de listas.

Respostas

1) Qual a sintaxe para criação de uma lista com e sem elementos?

R: A sintaxe para criação de uma lista é:

```
nome-da-lista = [ ]
```

Caso deseje inicializar a lista com elementos no momento de sua criação, deve-se informá-los entre os colchetes e separá-los por vírgula.

2) Os conceitos de elemento, tamanho da lista e índice fazem parte das listas. O que cada um deles significa?

R: Um elemento representa um dado guardado em um contêiner. O tamanho da lista representa a quantidade de elementos que a mesma possui. Por fim, o índice é um número que indica a posição em que um determinado elemento ocupa na lista, lembrando que o primeiro elemento está na posição 0.

3) Crie uma lista formada somente por vogais.

```
vogais = [ "a", "e", "i", "o", "u" ]
```

4) Crie uma lista formada por números ímpares de 0 a 100. Para isso utilize o conceito de compreensão de listas.

```
numeros = [ numero for numero in range(100) if numero %2 != 0]
```

10.4 Tuplas

Assim como as listas, as tuplas armazenam uma coleção de dados e os organizam em índices. No entanto, sua principal diferença é a **imutabilidade**, ou seja, após a sua criação não é possível alterar os seus elementos (adicionar ou remover), apenas realizar a leitura.

Você pode estar se perguntando por que necessitaria de um contêiner cujos elementos não é possível alterar. A seguir exemplificamos algumas situações em que seus dados, normalmente, não se alteram:

- Os vencedores do Óscar de melhor filme até o ano de 2019;
- Os nomes dos seus professores do 1º ano.
- As compras realizadas pelos clientes do supermercado no ano anterior;

Como já conhecemos os valores para cada situação e sabemos que eles não serão alterados, podemos utilizar tuplas. Na realidade, **há casos em que os dados não podem ser alterados**, apenas lidos. Por exemplo, deseja-se calcular a média das notas de uma turma do ano anterior que estão guardadas em um banco de dados. Neste caso, os dados não podem ser modificados para não comprometer um acontecimento passado. Assim, eles apenas serão recuperados dessa base de dados para que a média possa ser calculada.

Um benefício das tuplas é que seu desempenho computacional é superior quando comparadas às listas. Isso se torna especialmente importante quando há o manuseio de um grande volume de dados. Por exemplo, os gestores de uma rede social podem analisar o padrão de curtidas dos seus usuários para encontrar gostos em comum. Quanto mais pessoas utilizando esse software, maior será o volume de dados a ser processado. Assim, o uso de tuplas pode representar um importante ganho de performance para realizar essa operação.

Existem três sintaxes para criação de tuplas:

```
nome-da-tupla = ( elementos )
```

A declaração da tupla é feita utilizando parênteses de abertura e fechamento (diferente das listas, que utilizam colchetes). Dentro deles são definidos os elementos separados por vírgulas. Uma diferença em comparação às listas é que os elementos de uma tupla **sempre** são informados durante a sua criação, pois elas são imutáveis e não poderão ser alteradas posteriormente.

Veja um exemplo de declaração de tupla a seguir, com os vencedores do Oscar de 2017 a 2021.

```
vencedores_oscar_filme_2017_2021 = ("Nomadland", "Parasita", "Green Book – Um Guia Para a Vida", "A forma da água", "Moonlight: Sob a Luz do Luar")
```

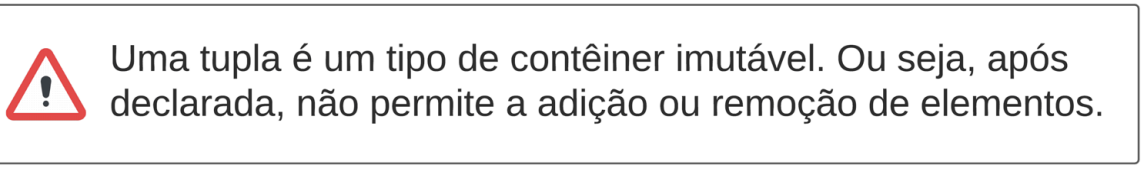

Figura 10.14: Uma tupla é um tipo de contêiner imutável. Ou seja, após declarada, não permite a adição ou remoção de elementos.

A segunda forma de criação de tuplas difere da anterior, pois não há o uso de parênteses na especificação dos elementos. Esse formato é conhecido por *empacotamento de tupla* ou do inglês *tuple packing*.

```
vencedores_oscar_filme_2017_2021 = "Nomadland", "Parasita", "Green Book – Um Guia Para a Vida", "A forma da água", "Moonlight: Sob a Luz do Luar"
```

Por fim, a terceira forma para criação de tuplas faz uso da função `tuple` e os elementos da tupla são informados como parâmetros. Detalhe para os parênteses duplos de abertura e fechamento.

```
vencedores_oscar_filme_2021_2017 = tuple(("Nomadland", "Parasita", "Green Book – Um Guia Para a Vida", "A forma da água", "Moonlight: Sob a Luz do
```

```
Luar"))
```

A instrução `tuple` também possibilita a conversão de uma lista em tupla:

```
vencedores_oscar_filme_2021_2017 = ["Nomadland", "Parasita", "Green Book – Um Guia Para a Vida", "A forma da água", "Moonlight: Sob a Luz do Luar"]
filmes_tupla = tuple(vencedores_oscar_filme_2021_2017)
```

Assim, a variável `filmes_tupla` será uma tupla com os elementos ("Nomadland", "Parasita", "Green Book – Um Guia Para a Vida", "A forma da água", "Moonlight: Sob a Luz do Luar").

O formato que será utilizado para criação das tuplas fica ao critério do desenvolvedor ou desenvolvedora.

Uma atenção especial deve ser dada para a criação de tuplas que possuem apenas um elemento, pois a sua sintaxe é diferente. Nesse caso é preciso incluir uma vírgula após a definição do valor:

```
oscar_2021 = ("Nomadland",)
oscar_2021 = "Nomadland",
```

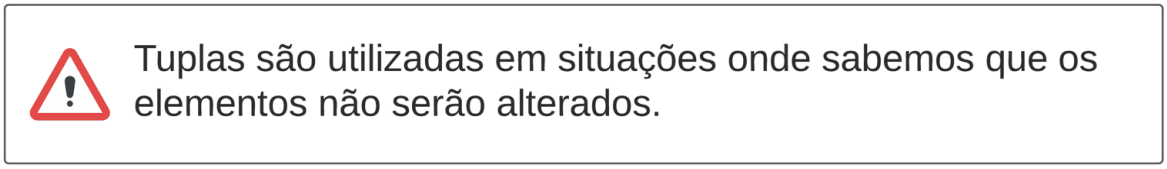

Figura 10.15: Tuplas são utilizadas em situações onde sabemos que os elementos não serão alterados.

Acesso aos elementos

Assim como as listas, as tuplas posicionam seus elementos em índices ordenáveis. O acesso aos elementos também é feito por meio de colchetes e informando o índice do elemento que queremos recuperar. Também se aplica tudo o que já foi apresentado, como o uso de índices negativos, `slice` , entre outros.

```
oscar_filme_2021_2017 = ("Nomadland", "Parasita", "Green Book – Um Guia Para a Vida", "A forma da água", "Moonlight: Sob a Luz do Luar")

melhor_filme_2021 = oscar_filme_2021_2017[0]
melhor_filme_2017 = oscar_filme_2021_2017[-1]
```

Como falamos, em razão da imutabilidade das tuplas elas não permitem a inserção ou remoção de elementos. O código a seguir exemplifica uma tentativa de modificação de um elemento da tupla:

```
oscar_filme_2021_2017 = ("Nomadland", "Parasita", "Green Book – Um Guia Para a Vida", "A forma da água", "Moonlight: Sob a Luz do Luar")
oscar_filme_2021_2017[0] = "Novo valor"
```

A execução do código ocasionará o erro: `TypeError: 'tuple' object does not support item assignment`

Iteração pelos seus elementos

A instrução `for` também é utilizada para iterar sobre os elementos de uma tupla, seguindo a mesma sintaxe utilizada com as listas:

```
oscar_filme_2021_2017 = ("Nomadland", "Parasita", "Green Book – Um Guia Para a Vida", "A forma da água", "Moonlight: Sob a Luz do Luar")

for filme in oscar_filme_2021_2017:
    print(filme)
```

Resultará na saída:

```
Nomadland
Parasita
Green Book – Um Guia Para a Vida
A forma da água
Moonlight: Sob a Luz do Luar
```

Operações em tuplas

As operações apresentadas para listas também se aplicam às tuplas, com exceção da função `sort`, pois o ordenamento da tupla não se

aplica uma vez que este contêiner é imutável e não permite o reposicionamento dos elementos. No entanto, o Python oferece uma alternativa por meio da função `sorted`, que recebe como parâmetro uma tupla e retorna uma lista com os elementos ordenados:

```
y = (2, 1, 4, 3, 4)
x = sorted(y)
print(x)
```

Assim, a variável *x* é uma lista com os elementos de *z* ordenados, apresentando o seguinte valor: `[1, 2, 3, 4, 4]`.

A função `sorted` também pode ser aplicada às listas e o benefício de utilizá-la é que a lista original não será alterada (lembra que o `sort` modifica a ordem dos elementos?):

```
z = [2, 1, 4, 3, 4]
x = sorted(z)
print(z)
print(x)
```

Sua execução resultará na saída:

```
[2, 1, 4, 3, 4]
[1, 2, 3, 4, 4]
```

Principais diferenças entre tuplas e listas

Muitos iniciantes confundem tuplas com listas e isso é natural. Há algumas diferenças que precisam ser consideradas na hora de escolher um contêiner ou outro. A primeira é avaliar se os elementos serão modificados ao longo do programa, pois nesses casos deve-se utilizar listas pela sua mutabilidade. Caso não exista essa necessidade, o Python oferece alguns indicativos que nos ajuda na escolha de qual contêiner utilizar:

- Para um conjunto de dados homogêneos, ou seja, quando são todos do mesmo tipo, as listas são recomendadas (ainda que

ela possa acomodar simultaneamente dados de tipos). Para dados de tipos heterogêneos as tuplas são indicadas;
- O uso de tuplas oferece uma proteção aos dados, pois há uma garantia de que eles não serão alterados;
- A criação e iteração de tuplas são procedimentos mais eficientes em razão da forma como o Python gerencia esse recurso na memória do computador. Ainda que na maioria dos casos você não vá se preocupar com esse aspecto, em situações onde há um grande volume de dados isso pode fazer diferença.

Questões

1) Qual a sintaxe para criação de uma tupla?

2) Crie uma tupla formada somente por vogais.

3) Quando utilizar uma tupla em vez de uma lista?

Respostas

1) Qual a sintaxe para criação de uma tupla?

R: Há três diferentes tipos de sintaxe para tuplas, sendo as principais:

```
nome-da-tupla = ( valores )
```

e

```
nome-da-tupla = valores
```

No segundo caso não são utilizados os parênteses, mas deve-se manter os dados separados por vírgula.

2) Crie uma tupla formada somente por vogais.

```
vogais = ( "a", "e", "i", "o", "u" )
```

3) Quando utilizar uma tupla ao invés de uma lista?

As tuplas devem ser usadas para dados que não serão alterados. Além disso, elas apresentam uma melhor performance em sua criação e iteração.

10.5 Erros comuns

Acreditar que os índices das listas e tuplas iniciam em 1. Um dos principais erros ao utilizar listas e tuplas e acreditar que seu índice inicia em um. Isso afeta não apenas o acesso ao primeiro elemento, como também ao último, pois este é representado pelo quantitativos de elementos subtraído de 1.

Código com problema:

```
lista = [5, 10, 7]
primeiro_elemento = lista[1]
```

Código corrigido:

```
lista = [5, 10, 7]
primeiro_elemento = lista[0]
```

Tentar acessar um índice inexistente. Os elementos pertencentes às listas e tuplas possuem um índice que inicia em zero e finaliza em n-1. Tentar acessar um índice fora desse alcance resulta em um erro.

Código com problema:

```
lista = [5, 10, 7]
print(lista[3]) # Tentativa de acessar o último elemento da lista
```

Código corrigido:

```
lista = [5, 10, 7]
print(lista[2]) # Tentativa de acessar o último elemento da lista
```

Utilizar parênteses para acessar um elemento. O acesso aos elementos de uma lista ou tupla é feito com colchetes. Muitos confundem e utilizam parêntesis.

Código com problema:

```
lista = [5, 10, 7]
elemento = lista(2)
```

Código corrigido:

```
lista = [5, 10, 7]
elemento = lista[2]
```

10.6 Resumo e mapa mental

- Neste capítulo aprendemos que os contêineres são utilizados para armazenar uma coleção de dados que possuem alguma relação entre si;
- Conhecemos quatro tipos de contêineres: listas, tuplas, conjuntos e dicionários. Cada um possui a sua própria forma de representar os dados que armazenam;
- Listas e tuplas posicionam seus elementos em índices numéricos que iniciam em 0 e vão até n-1;
- Listas apresentam a característica de mutabilidade, ou seja, podem ter seus elementos alterados após a sua criação, enquanto as tuplas são imutáveis;
- Tuplas apresentam uma melhor eficiência em operações de iteração, quando comparadas às listas e por essa razão são recomendadas para armazenar grandes volumes de dados.

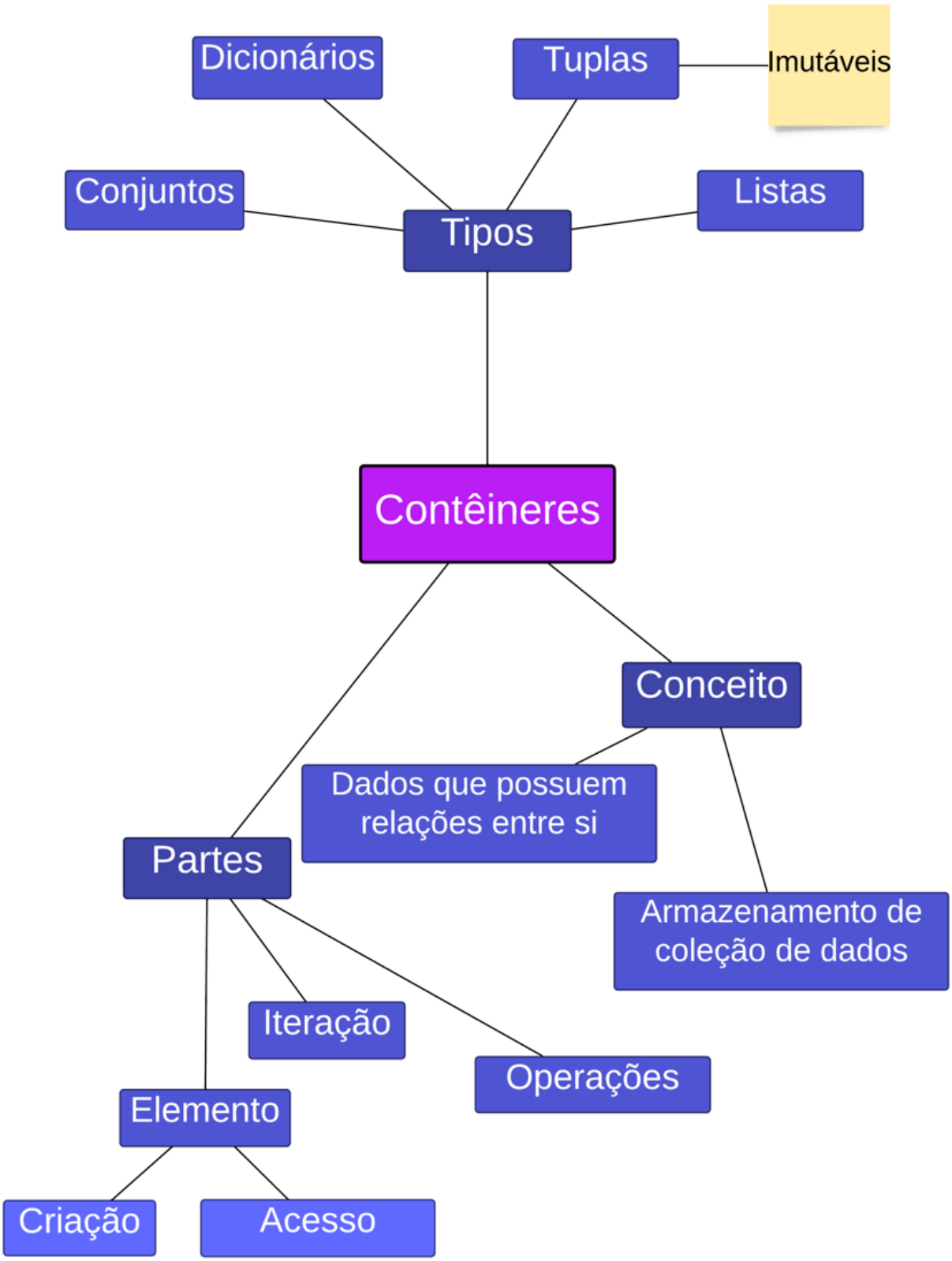
Dicionários
Tuplas
Imutáveis
Conjuntos
Tipos
Listas
Contêineres
Conceito
Dados que possuem relações entre si
Partes
Armazenamento de coleção de dados
Iteração
Operações
Elemento
Criação
Acesso

Figura 10.16: Mapa mental do capítulo 10.

CAPÍTULO 11
Conjuntos, Dicionários e Matrizes

1. O que você já sabe?

Os contêineres possibilitam o armazenamento de um grupo de dados em uma única variável. Eles são importantes em situações em que esses dados possuem alguma relação entre si e são manipulados conjuntamente. Aprendemos que há diferentes tipos de contêineres, cada qual com uma característica em particular e adequado para determinadas situações.

Conhecemos os contêineres do tipo lista e tupla, que atribuem índices aos dados armazenados. Esses índices são utilizados para recuperação do elemento armazenado e, no caso das listas, também possibilitam a sua alteração.

2. O que veremos?

Neste capítulo vamos abordar outros dois dos contêineres oferecidos pelo próprio Python: os conjuntos e dicionários, além de apresentar uma forma mais complexa de uso das listas, conhecida por matriz ou lista aninhada.

3. Por que é importante?

Conjuntos e dicionários oferecem novas operações que podemos realizar nos conjuntos de dados, bem como um formato diferente para representação de dados. Há situações em que o uso desses contêineres é favorável, em comparação às listas e tuplas, e vamos discutir estes fatores neste capítulo.

O que você vai aprender?

Ao término deste capítulo você compreenderá:

- O que são os contêineres do tipo conjunto e dicionário;
- As situações que justificam o uso dos contêineres apresentados;
- A sintaxe e operações disponíveis em cada um destes contêineres;
- A representação e uso de matrizes no Python.

11.1 Novos formatos de contêineres para representação de dados

Os contêineres apresentados neste capítulo possibilitam novas formas de representação e manipulação de dados no Python. Isso é importante, pois sabemos que os softwares resolvem problemas do mundo real e que muitas vezes simples variáveis (ou mesmo listas e tuplas) são limitadas em sua representação.

O conhecimento deste capítulo somado ao anterior possibilita ao programador(a) fazer uma melhor avaliação sobre qual tipo de contêiner deve utilizar em seu software. Listas e tuplas se mostram adequadas para armazenar sequências de dados, mas não dispõem de certas operações que podem ser realizadas entre contêineres. Por exemplo, como faríamos para identificar os elementos repetidos em duas listas diferentes? Uma possibilidade é apresentada no código a seguir:

```
lista_um = [1,2,3,4]
lista_dois = [3,4,5,6]
elementos_repetidos = []

for dado in lista_um:
    if dado in lista_dois:
        elementos_repetidos.append(dado)

print(elementos_repetidos)
```

A saída do código será: `[3, 4]`.

A verificação da duplicidade somente foi possível, pois implementamos essa funcionalidade. No entanto, existem outros contêineres que realizam essa ação de uma forma mais simplificada e pronta para uso.

Uma outra limitação das listas e tuplas é que os elementos armazenados não carregam consigo uma "semântica" que pode ser útil para indicar do que se trata um determinado dado. Vamos ilustrar esse problema, no código a seguir temos uma lista formada pelo preço de três produtos em um supermercado:

```
precos_supermercado = [ 4.5, 5, 3.5 ]
```

Do que se tratam os preços acima? Feijão, arroz, macarrão? Cebola, tomate e alho? Não é possível saber. Veremos neste capítulo que os conjuntos e dicionários resolvem parte destes problemas. Também conheceremos as matrizes que são de igual importância e permitem um outro tipo de representação para nossos dados.

11.2 Conjuntos

Este tipo de contêiner apresenta uma forte similaridade com os conjuntos matemáticos e permite realizar operações de união, intersecção e diferença. Isso é importante pois possibilita encontrar semelhanças ou diferenças entre os elementos de dois ou mais conjuntos. Lembra do exemplo anterior onde identificamos os elementos que se repetem entre dois contêineres? Apresentaremos a sua reescrita com o uso de conjuntos e não se preocupe com a sintaxe neste momento, pois em breve ela será apresentada.

```
conjunto_um = {1,2,3,4}
conjunto_dois = {3,4,5,6}
elementos_repetidos = conjunto_um & conjunto_dois
```

Observe como realizamos a mesma operação com apenas três linhas de código.

Ao longo desta seção utilizaremos como exemplo uma empresa que possui dois escritórios, nomeados A e B. Deseja-se manter um registro sobre os trabalhadores que são representados pelos seus números de CPFs. Os conjuntos serão utilizados para armazenar os dados.

O grande benefício no uso deste contêiner são as operações que podem ser realizadas. Por exemplo, podemos identificar os funcionários que estão alocados nos dois escritórios ao mesmo tempo. Isso é possível por meio de uma operação de intersecção entre os conjuntos, como apresentado na figura a seguir.

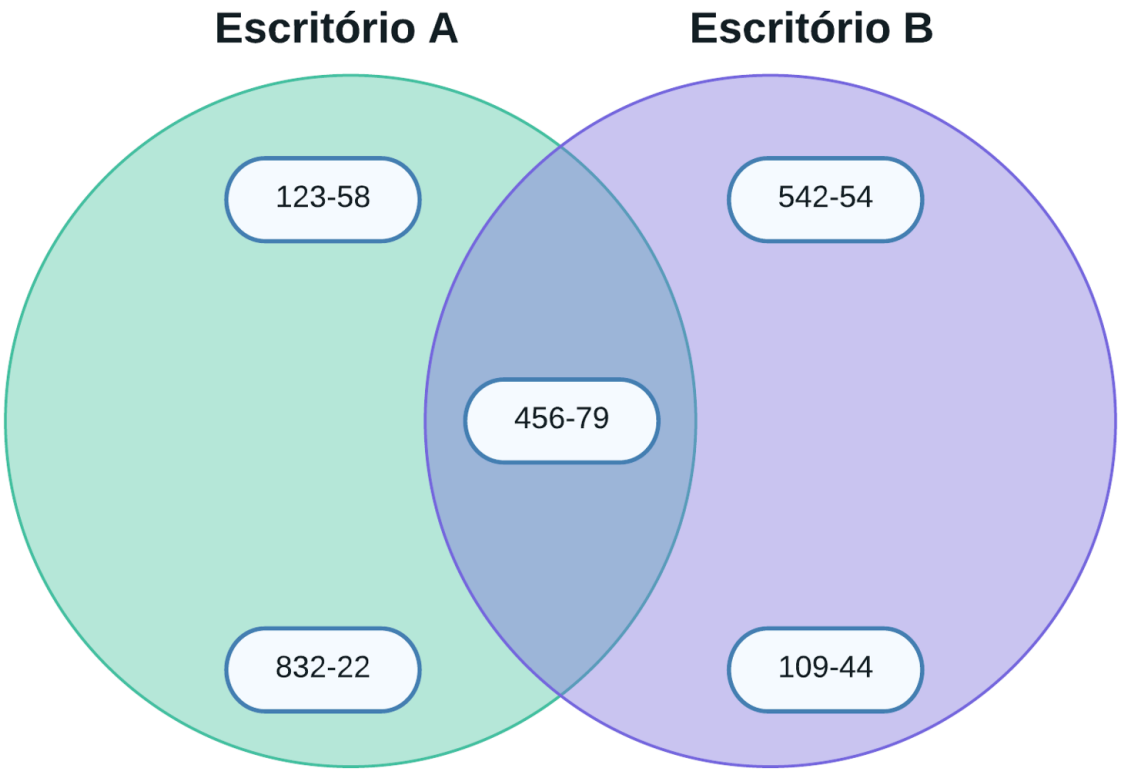

Figura 11.1: Conjuntos com os funcionários dos escritórios.

Há duas sintaxes para criação de conjuntos. A primeira segue o seguinte formato:

nome-do-conjunto = { elementos }

Por exemplo:

```
cpf_empresa_a = { 12358, 83222, 45679 }
cpf_empresa_b = { 54254, 10944, 45679 }
```

No entanto, com a sintaxe apresentada não é possível criar um conjunto sem elementos. Para isso, utiliza-se a função `set`:

```
cpf_empresa_a = set()
```

Posteriormente, os elementos poderão ser incluídos no conjunto.

A função `set` também permite criar um conjunto a partir de um iterável, como listas, tuplas, range, *Strings*, entre outros. Nesse caso, cada elemento do iterável será um elemento do conjunto. O código a seguir demonstra o uso dessa função a partir de uma instrução `range(5)` que cria uma sequência numérica de 0 a 4.

```
numeros = set(range(5))
print(numeros)
```

Esse código resultará no conjunto: `{ 0, 1, 2, 3, 4 }`.

Um conjunto pode ser inicializado com os dados de uma lista:

```
letras = set(["a", "b", "c"])
print(letras)
```

Resultará no conjunto: `{ "a", "c", "b" }`.

Por fim, é possível utilizar uma *String* como parâmetro de `set`:

```
palavra = set("bola")
print(palavra)
```

A execução do código resultará no conjunto: `{ "a", "b", "o", "l" }` (em seu computador as letras podem estar dispostas de outra forma). Esse código apresenta duas situações interessantes. A primeira é que a *String* passada como parâmetro do `set` será dividida em letras, e cada uma delas serão alocadas no conjunto. O segundo fato relevante é que o conjunto produzido não guardará as letras em ordem alfabética, pois os elementos de um conjunto não são armazenados na ordem em que são informados na sua criação. Essa informação merece atenção, pois muitos acreditam que os conjuntos possuem um índice numérico, como as listas e tuplas.

Por fim, os conjuntos não permitem elementos duplicados. Essa característica é útil, pois existem situações onde não é permitida a existência de dois dados iguais. Por exemplo, o registro sobre os funcionários de um escritório não pode ter dados duplicados, pois o CPF é único para cada cidadão brasileiro.

A garantia oferecida pelos conjuntos de que em seus elementos não há duplicatas é especialmente útil, pois sem ela seria preciso que o(a) programador(a) implementasse um controle sobre duplicatas, o que demandaria um certo trabalho.

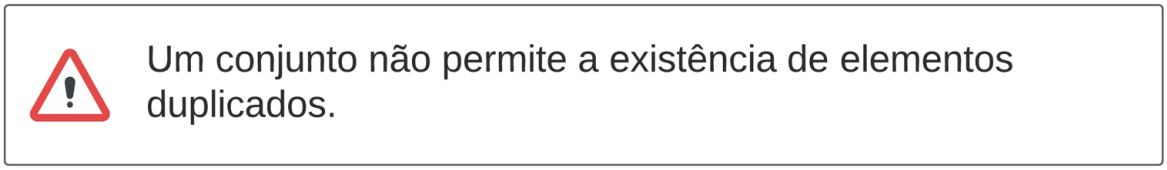

Figura 11.2: Um conjunto não permite a existência de elementos duplicados.

Acesso e modificação dos elementos

Os elementos de um conjunto não podem ser acessados individualmente, pois não há um índice para eles. Portanto, a única forma de recuperá-los é iterando o conjunto por meio de um `for`. No exemplo a seguir, apresenta-se essa operação seguida da impressão de cada elemento contido nele:

```
cpf_empresa_b = { 54254, 10944, 45679 }
for cpf in cpf_empresa_b:
    print(cpf)
```

Não é possível modificar um elemento. Para alterar um dado do conjunto é preciso excluí-lo e em seguida inserir um novo dado.

Inserção de elementos

A inserção de elementos em um conjunto é feita por meio das funções `add` e `update`. Elas diferem na quantidade de dados que podem ser inseridos com uma única instrução.

Com a função `add` é possível inserir apenas um elemento. Sua sintaxe é apresentada a seguir:

```
nome-do-conjunto.add( elemento )
```

A função recebe um único parâmetro que é o elemento a ser adicionado. A seguir, apresenta-se o seu uso com a criação de um conjunto com três elementos, seguido da adição de um novo dado.

```
cpf_empresa_b = { 54254, 10944, 45679 }
cpf_empresa_b.add( 75342 )
print(cpf_empresa_b)
```

A execução do código resultará na saída: `{10944, 75342, 54254, 45679}`.

o Python não acusará nenhum erro caso tente inserir um elemento que já existe no conjunto. No entanto, nada vai acontecer, pois não são permitidos dados duplicados.

A função `update` possibilita a inserção de múltiplos elementos de uma única vez em um conjunto. O parâmetro deve ser um iterável, como tuplas, listas, strings ou mesmo outros conjuntos.

nome-do-conjunto

A seguir, apresenta-se o uso do `update` para adicionar os dados de uma lista em um conjunto:

```
novos_funcionarios = { 75342, 85973 }
cpf_empresa_b = { 54254, 10944, 45679 }
cpf_empresa_b.update( novos_funcionarios )
print(cpf_empresa_b)
```

O conjunto apresentará os elementos: `{10944, 75342, 54254, 45679, 85973}`.

A função `update` também recebe um número variado de parâmetros. Assim, é possível adicionar os dados de múltiplos iteráveis com uma única execução.

nome-do-conjunto.update(iteravel_a, iteravel_b, ..., iteravel_n)

No código a seguir temos os dados de novos funcionários que serão admitidos em um dos escritórios. Os CPFs dessas pessoas estão em dois conjuntos diferentes: `novos_funcionarios_um` e `novos_funcionarios_dois`.

```
novos_funcionarios_um = { 75342, 85973 }
novos_funcionarios_dois = { 12365, 98754 }
cpf_empresa_b = { 54254, 10944, 45679 }
cpf_empresa_b.update( novos_funcionarios_um, novos_funcionarios_dois )
print(cpf_empresa_b)
```

Observe como a função `update` recebe dois conjuntos como parâmetros. Esses elementos serão inclusos no o conjunto `cpf_empresa_b` que apresentará os seguintes elementos: `{12365, 10944, 98754, 75342, 54254, 45679, 85973}`.

A passagem de parâmetro de um dado não iterável ao `update` ocasionará em um erro:

```
cpf_empresa_b = { 54254, 10944, 45679 }
cpf_empresa_b.update( 75342 )
```

O resultado da execução deste código é uma mensagem de erro do Python: `TypeError: 'int' object is not iterable`. Está escrito que o elemento do tipo int (número inteiro) passado como parâmetro não é iterável.

Remoção de elementos

A exclusão de elementos de um conjunto é feita por meio das funções `remove`, `discard` e `clear`, que apresentam diferentes comportamentos.

A função `remove` exclui um elemento do conjunto. Ela recebe como parâmetro o dado que deve ser excluído e sua sintaxe é representada pelo formato:

```
nome-do-conjunto.remove( elemento )
```

Exemplo de uso da função:

```
cpf_empresa_b = { 54254, 10944, 45679 }
cpf_empresa_b.remove( 45679 )
print(cpf_empresa_b)
```

O conjunto final será formado pelos elementos: `{10944, 54254}`.

Caso o elemento informado como parâmetro de `remove` não exista, o Python acusará uma mensagem de erro:

```
cpf_empresa_b = { 54254, 10944, 45679 }
cpf_empresa_b.remove( 12345 )
print(cpf_empresa_b)
```

O seguinte erro será ocasionado ao executar o código: `KeyError: 12345`.

É importante entender o que ocorre com a execução do código quando erros são produzidos. Observe que na terceira linha temos uma instrução `print`. No entanto, ela não será executada, pois o Python interrompe o programa, fazendo com que as instruções após a linha que ocasionou o problema não sejam executadas. Vamos abordar este tema de erros no capítulo 12.

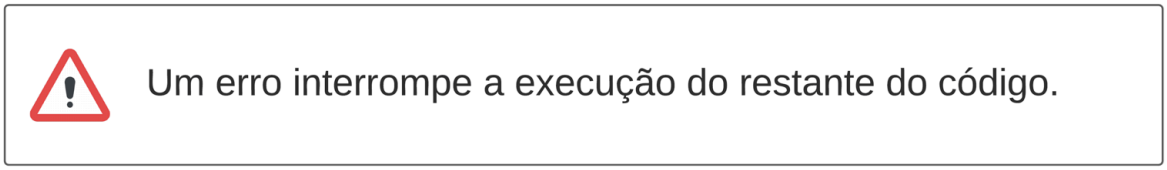

Figura 11.3: Um erro interrompe a execução do restante do código.

Devemos utilizar a função `discard` para evitar que um *KeyError* seja lançado ao tentar remover um elemento inexistente. Sua sintaxe é a seguinte:

```
nome-do-conjunto.discard( elemento )
```

Nenhum erro será ocasionado ao executar o código a seguir:

```
cpf_empresa_b = { 54254, 10944, 45679 }
cpf_empresa_b.discard( 12345 )
print(cpf_empresa_b)
```

A saída será um conjunto formado pelos elementos: `{ 54254, 10944, 45679 }`.

Para finalizar, a função `clear` é utilizada para remover todos os elementos do conjunto. Sua sintaxe é representada por:

```
nome-do-conjunto.clear()
```

Um exemplo de uso desta função é ilustrado na sequência:

```
cpf_empresa_b = { 54254, 10944, 45679 }
cpf_empresa_b.clear()
print(cpf_empresa_b)
```

Após o código ser executado a variável `cpf_empresa_b` será um conjunto vazio.

Operações em conjuntos

O grande benefício dos conjuntos são as operações que podem ser realizadas. Assim, é possível realizar a interseção entre conjuntos, encontrar as diferenças, como também realizar a união de um ou mais conjuntos. Há duas formas em que essas ações podem ser realizadas, por meio de *funções* ou *operadores*. Vamos apresentá-las a seguir.

União. Essa operação cria um conjunto formado pela junção dos elementos de dois ou mais conjuntos. É possível realizar essa ação por meio do operador `|`, como representado na sintaxe:

```
resultado_uniao conjunto_a | conjunto_b
```

No exemplo a seguir vamos criar um conjunto com todos os funcionários de uma empresa. Isso será realizado pela união entre os conjuntos que representam os funcionários de cada escritório.

```
cpf_empresa_a = { 12358, 83222, 45679 }
cpf_empresa_b = { 54254, 10944, 45679 }

todos_funcionarios = cpf_empresa_a | cpf_empresa_b

print(todos_funcionarios)
```

O resultado é um conjunto formado pelos elementos `{10944, 12358, 83222, 54254, 45679}` e armazenado na variável `todos_funcionarios`. Observe que o elemento 45679 é comum aos dois conjuntos e somente será inserido uma vez no novo conjunto. Isso ocorre porque os conjuntos não permitem dados duplicados.

A segunda forma de realizar a operação de união é por meio da função `union` que recebe como parâmetros um ou mais conjuntos. Assim, os elementos do conjunto onde a função foi chamada serão unificados aos elementos dos conjuntos passados como parâmetro.

```
resultado_uniao = conjunto_a.union( conjunto_b )
```

Exemplifica-se a seguir o uso dessa função:

```
cpf_empresa_a = { 12358, 83222, 45679 }
cpf_empresa_b = { 54254, 10944, 45679 }

todos_funcionarios = cpf_empresa_a.union( cpf_empresa_b )

print(todos_funcionarios)
```

A execução deste código fará com que a variável `todos_funcionarios` possua como elementos os dados contidos em `cpf_empresa_a` e `cpf_empresa_b`.

Também é possível realizar a união com mais de um conjunto. Neste caso, a sintaxe segue o seguinte formato:

```
resultado_uniao = conjunto_a | conjunto_b | conjunto_c | ... | conjunto_n

resultado_uniao = conjunto_a.union( conjunto_b, conjunto_c, ..., conjunto_n )
```

O código a seguir demonstra o uso dessas duas instruções:

```
cpf_empresa_a = { 12358, 83222, 45679 }
cpf_empresa_b = { 54254, 10944, 45679 }
cpf_empresa_c = { 159753, 789457 }

todos_funcionarios = cpf_empresa_a | cpf_empresa_b | cpf_empresa_c

todos_funcionarios = cpf_empresa_a.union( cpf_empresa_b, cpf_empresa_c )

print(todos_funcionarios)
```

Você pode estar se perguntando quando utilizar o operador `|` ou a função `union`. Se o objetivo for unir dois ou mais conjuntos, não importa qual será utilizada.

A função `union` também permite a união de conjuntos com dados de outros tipos, desde que sejam iteráveis, como listas, tuplas, range, entre outras. O código a seguir demonstra a união entre um conjunto com uma lista:

```
cpf_empresa_a = { 12358, 83222, 45679 }
lista = [ "a", "b", "c" ]

uniao = cpf_empresa_a.union(lista)
print(uniao)
```

A execução do código resultará na saída: `{'a', 12358, 'b', 45679, 83222, 'c'}`

Interseção. Consiste na identificação dos elementos que se repetem entre dois ou mais conjuntos. Utiliza-se o operador `&` ou a função `intersection`:

```
resultado_intersecao = conjunto_a & conjunto_b

resultado_intersecao = conjunto_a.intersection( conjunto_b )
```

O código a seguir demonstra a realização da operação de interseção:

```
cpf_empresa_a = { 12358, 83222, 45679 }
cpf_empresa_b = { 54254, 10944, 45679 }

comuns_as_empresas = cpf_empresa_a & cpf_empresa_b

comuns_as_empresas = cpf_empresa_a.intersection( cpf_empresa_b )
```

A variável `comuns_as_empresas` resultará em um conjunto com o elemento `45679`, pois é o que se repete entre os conjuntos.

Diferença. Identifica os elementos que existem em um conjunto, mas não no outro. Para executar essa operação, utiliza-se o operador `-` ou a função `difference`:

```
resultado_diferenca = conjunto_a - conjunto_b

resultado_diferenca = conjunto_a.difference( conjunto_b )
```

Um exemplo dessas operações é apresentado a seguir:

```
cpf_empresa_a = { 12358, 83222, 45679 }
cpf_empresa_b = { 54254, 10944, 45679 }

exclusivos_empresa_a = cpf_empresa_a - cpf_empresa_b
exclusivos_empresa_a = cpf_empresa_a.difference( cpf_empresa_b )

exclusivos_empresa_b = cpf_empresa_b - cpf_empresa_a
exclusivos_empresa_b = cpf_empresa_b.difference( cpf_empresa_a )
```

A ordem em que os conjuntos são informados na operação modifica o resultado. Por exemplo, a instrução a seguir verificará quais funcionários pertencem somente ao conjunto `cpf_empresa_a` quando comparado ao conjunto `cpf_empresa_b`.

```
resultado = cpf_empresa_a - cpf_empresa_b
```

Por outro lado, a próxima instrução verificará quais funcionários pertencem somente ao conjunto `cpf_empresa_b` quando comparado ao conjunto `cpf_empresa_a`.

```
resultado = cpf_empresa_b - cpf_empresa_b
```

Questões

1) Qual a sintaxe para criação de um conjunto?

2) Crie um conjunto formado somente por vogais.

3) Quando utilizar um conjunto em vez de listas e tuplas?

Respostas

1) Qual a sintaxe para criação de um conjunto?

R: Os conjuntos podem ser criados diretamente pela sintaxe:

```
nome-do-conjunto = { elementos }
```

Ou por meio da função `set`, que aceita como parâmetro qualquer instrução iterável, como listas, tuplas, entre outros.

2) Crie um conjunto formado somente por vogais.

```
vogais = { "a", "e", "i", "o", "u" }
```

3) Quando utilizar um conjunto em vez de listas e tuplas?

R: Os conjuntos são úteis quando se deseja realizar operações, como união, intersecção e diferença, entre contêineres. Outro benefício dos conjuntos é a restrição que possuem para dados duplicados. Assim, o(a) programador(a) não precisa implementar essa verificação, pois por padrão os conjuntos são livres de duplicatas.

11.3 Dicionários

Para compreender a importância dos dicionários, primeiro é preciso entender uma limitação que listas e tuplas apresentam. Aprendemos que esses dois contêineres organizam seus elementos por meio de índices numéricos. Por exemplo, na imagem a seguir há um conjunto de dez notas armazenadas em uma lista.

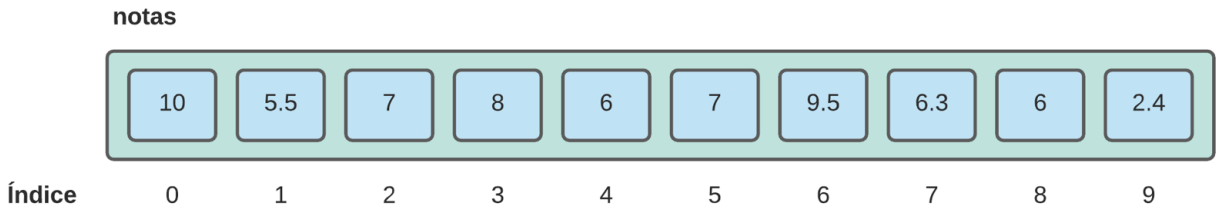

Figura 11.4: Representação de uma lista com seus índices.

Observe que não temos nenhuma informação sobre as pessoas que alcançaram essas notas. Assim, como fazemos para saber a qual aluno pertence a nota 10 e o 9.5? Da forma como o dado está representado não é possível saber.

O problema apresentado pode ser resolvido com um dicionário. Seus elementos são organizados por meio de índices que podem ser textuais (nomeados de chaves). Vamos explicar, enquanto as listas e tuplas associam a cada elemento um índice numérico (figura anterior), os dicionários podem associar informações textuais. Assim, podemos criar índices com os nomes dos alunos e para cada um deles associar a sua nota, como pode ser visto na figura a seguir.

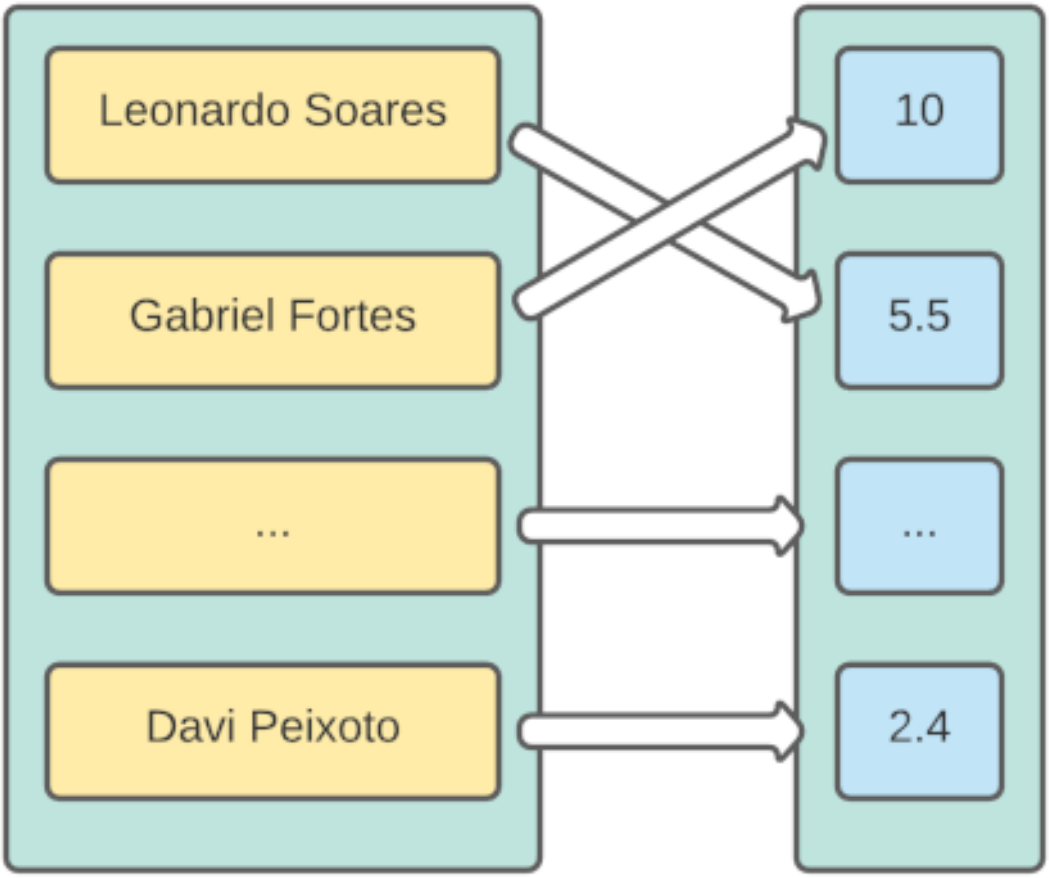

Figura 11.5: Representação de um dicionário.

Percebe-se que **os elementos de um dicionário são formados por duas partes, uma chave e um valor**. A chave, que seria o equivalente ao índice, serve como uma etiqueta que indica como um determinado dado pode ser acessado. O valor por sua vez pode ser qualquer dado válido para o Python.

A forma como os elementos são estruturados (`chave => valor`) assemelha-se a um dicionário de palavras. Por exemplo, buscamos por uma palavra (que analogamente seria a chave) para saber o seu significado (equivalente ao nosso valor). Por isso esse contêiner recebe esse nome.

A sintaxe para criação de um dicionário segue o formato:

```
nome-dicionario = { chave : valor, chave : valor, ... chave: valor }
```

Observe que quando vamos informar um dado no dicionário é preciso informar a chave e o seu valor. A chave deve ser uma *String*, número ou tupla (essa última deve seguir os critérios descritos na documentação oficial (https://docs.python.org). A chave de um dicionário deve ser um valor único, não sendo possível existir duas iguais. O valor pode ser de qualquer dado suportado pelo Python e não há restrições em relação a valores iguais. Por fim, os elementos de um dicionário devem ser separados por vírgulas.

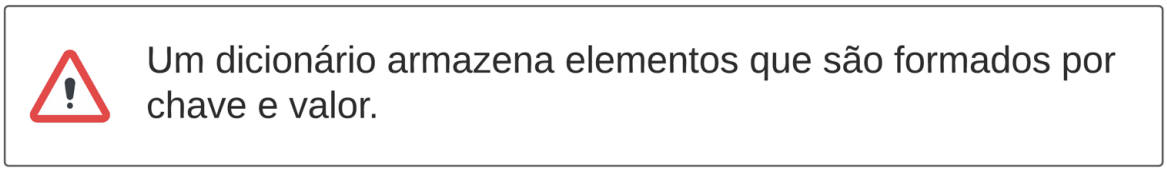

Figura 11.6: Um dicionário armazena elementos que são formados por chave e valor.

No código a seguir será criado um dicionário para representar as notas dos alunos com os seus nomes como chaves:

```
notas = { "LeonardoSoares" : 5.5, "GabrielFortes" : 10, "LinusTorvalds" : 8, "SteveWozniak":3 }
```

Observe que a chave `LeonardoSoares` associa-se ao valor 5.5, `GabrielFortes` ao valor 10, e assim por diante.

As chaves também podem ser dados numéricos. A seguir criamos um dicionário para guardar a tabela ASCII, um padrão americano para representação de letras, algarismos e outros sinais, em computadores. Nesta tabela a letra `a` é representada pelo número 97, enquanto a letra `c` é representada pelo número 99:

```
tabela_ascii = { 97: "a", 99: "c" }
```

Observe que os números não são informados entre aspas.

Os valores guardados por um dicionário podem ser de qualquer tipo válido para o Python. A seguir ilustramos um dicionário que tem como elemento uma lista:

```
dicionario = {"numeros" : [ 1, 2, 3] }
```

No código temos a chave `numeros` que guarda um dado do tipo lista.

Dicionários também podem ser criados por meio da função `dict`, que recebe como parâmetro uma combinação de chaves e valores, representado pela sintaxe:

> nome-dicionario = dict({ chave : valor, chave : valor, … chave: valor })

Exemplificada no código:

```
notas = dict({ "LeonardoSoares" : 5.5, "GabrielFortes" : 10,
"LinusTorvalds" : 8, "SteveWozniak":3 })
```

Há outras formas de criação de dicionários por meio da função `dict`. Quando a chave for uma *String*, pode-se utilizar a sintaxe:

> nome-dicionario = dict(chave = valor, chave = valor, …, chave = valor)

Essa sintaxe apresenta algumas diferenças em relação às anteriores. As chaves não são informadas entre aspas e a separação da chave e valor ocorre por um sinal de igualdade. Vamos ver um exemplo:

```
notas = dict(LeonardoSoares = 5.5, GabrielFortes = 10, LinusTorvalds = 8,
SteveWozniak = 3)
```

A função `dict` também pode receber como parâmetro uma lista de tuplas. Vamos relembrar os conceitos de listas e tuplas a partir do

exemplo a seguir. Nele, vamos declarar uma lista e uma tupla. Por fim, teremos uma lista formada por tuplas.

```
lista = [1, 2, 3, 4]
tupla = ("a", "b", "c")
lista_tuplas = [ ("a", 1), ("b", 2), ("c", 3) ]
```

Observe que a variável `lista_tuplas` é uma lista e cada um dos seus elementos são tuplas. Mas atenção, para utilizar esse tipo de lista na função `dict`, **cada tupla deve ser formada por dois elementos**. O primeiro será a chave e o segundo, o valor. Ilustramos no exemplo a seguir:

```
notas = dict([ ("LeonardoSoares", 5.5), ("GabrielFortes", 10),
("LinusTorvalds", 8), ("SteveWozniak",3) ])
```

Neste código, *LeonardoSoares* será a chave do valor *5.5*.

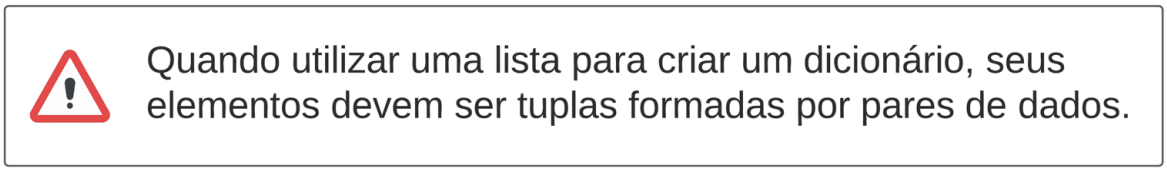

Figura 11.7: Quando utilizar uma lista para criar um dicionário, seus elementos devem ser tuplas formadas por pares de dados.

Para criar um dicionário sem elementos pode-se utilizar a seguinte sintaxe:

```
meu_dicionario = { }
meu_dicionario = dict()
```

Os dicionários também possuem o seu conceito de compreensão de dicionários (*dictionary comprehension*). Assim, é possível inicializar seus valores a partir de uma instrução de programação. A sintaxe é representada da seguinte forma:

```
nome-do-dicionario = { item : expressão for item in iterador }
```

Esses dados serão gerados por meio de uma repetição em um iterador. Em cada iteração um elemento é criado, onde a variável `item` será a chave e o seu valor será o resultado de `expressão`. Por exemplo, para criar um dicionário com quatro chaves numéricas, de 0 a 3, e com os valores sendo estes números multiplicados por dois, utiliza-se o seguinte código:

```
dicionario = {x: x*2 for x in range(4)}
print(dicionario)
```

A saída do código é: `{0: 0, 1: 2, 2: 4, 3: 6}`

Acesso e modificação dos elementos

O acesso de um elemento do dicionário é feito por meio da sua chave e segue a sintaxe:

```
valor = nome-do-dicionario[ chave ]
```

A variável `valor` vai armazenar o valor associado à chave do elemento especificado. No exemplo a seguir criamos um dicionário de notas e acessamos a nota do estudante de chave `LeonardoSoares`.

```
notas = dict(LeonardoSoares = 5.5, GabrielFortes = 10, LinusTorvalds = 8,
SteveWozniak = 3)
valor_guardado = notas["LeonardoSoares"]
print(valor_guardado)
```

Com a execução deste código, a variável `valor_guardado` armazena o valor 5.5.

Para acessar um valor que tem por chave um número devemos informá-lo sem aspas:

```
tabela_ascii = { 97: "a", 99: "c" }
print(tabela_ascii[97])
```

Quando um dicionário armazena uma lista ou tupla, o acesso aos elementos guardado por eles é um pouco diferente:

valor = nome-do-dicionario[chave][índice]

No código a seguir ilustramos a aplicação da sintaxe anterior:

```
dicionario = {"numeros" : [ 1, 2, 3] }
segundo_elemento_numeros = dicionario["numeros"][1]
```

Com o código, a variável `segundo_elemento_numeros` vai guardar o número 2 que é o segundo elemento da lista representada pela chave `numeros`.

A tentativa de acessar uma chave que não existe ocasionará um erro do tipo `KeyError`:

```
notas = dict(LeonardoSoares = 5.5, GabrielFortes = 10, LinusTorvalds = 8,
SteveWozniak = 3)
print( notas["GuidoRossum"] )
```

Isso resultará no erro `KeyError: 'GuidoRossum'`.

Para evitar erros como o apresentado, é possível verificar a existência da chave no dicionário antes do seu uso. Essa ação pode ser realizada por meio do operador `in`, que resultará em `True` quando a chave existir ou `False` em caso contrário. No código a seguir, ilustramos esse conceito para verificar se a chave `GuidoRossum` está presente no dicionário. Essa ação é feita por meio de uma condição e o seu escopo resultará na impressão da nota deste estudante. Entretanto, se não houver essa nota, então nada seria impresso na tela:

```
notas = dict(LeonardoSoares = 5.5, GabrielFortes = 10, LinusTorvalds = 8,
SteveWozniak = 3)
```

```
if "GuidoRossum" in notas:
    print( notas["GuidoRossum"] )
```

Outra forma de acessar um elemento é por meio da função `get`. Essa função recebe como parâmetro uma chave e retornará o valor associado a ela.

```
notas = dict(LeonardoSoares = 5.5, GabrielFortes = 10, LinusTorvalds = 8,
SteveWozniak = 3)
valor_guardado = notas.get("LeonardoSoares")
print(valor_guardado)
```

A variável `valor_guardado` armazena o dado 5.5. Uma diferença desta sintaxe em comparação ao uso de colchetes apresentado anteriormente é que na inexistência da chave informada como parâmetro de `get`, não será lançado o erro `KeyError`. Nesse caso, a função retornará um valor do tipo `None`:

```
notas = dict(LeonardoSoares = 5.5, GabrielFortes = 10, LinusTorvalds = 8,
SteveWozniak = 3)

notas_guido = notas.get("GuidoRossum")
print( notas_guido )
```

Isso apresentará como saída: `None`. Para o Python, essa informação representa literalmente "nada". Pode parecer estranho, mas por vezes na programação precisamos deste recurso para representar situações como essa, onde a informação representa um vazio. Em outras linguagens de programação esse recurso seria o equivalente ao *null*. Abordaremos este tema em detalhes no capítulo 12.

Antes de avançarmos considere o código a seguir, onde recuperamos um valor do dicionário de notas e o guardamos em uma variável:

```
notas = dict(LeonardoSoares = 5.5, GabrielFortes = 10, LinusTorvalds = 8,
SteveWozniak = 3)
valor_guardado = notas["LeonardoSoares"]
```

É importante compreender que o valor armazenado na variável `valor_guardado` não possui relação com o valor guardado por `notas["LeonardoSoares"]`. Isso significa que a alteração em um não implica uma mudança no outro, como exemplificado a seguir:

```
notas = dict(LeonardoSoares = 5.5, GabrielFortes = 10, LinusTorvalds = 8,
SteveWozniak = 3)
valor_guardado = notas["LeonardoSoares"]
print(valor_guardado)
valor_guardado = 8
print(valor_guardado)
print(notas["LeonardoSoares"])
```

O código possui três saídas. Na primeira, estamos exibindo o valor da variável `valor_guardado`, que é o dado armazenado pela chave `LeonardoSoares`, resultando em `5.5`. Após isso, alteramos o valor da variável `valor_guardado` para 8, consequentemente o `print` apresentará na tela `8`. Por fim, apresentamos o valor do elemento `notas["LeonardoSoares"]` que é `5.5`, pois ele não foi alterado pelas mudanças na variável `valor_guardado`.

A falta de ligação entre as variáveis ocorre, pois elas estão em **posições diferentes da memória do computador**. Falamos sobre isso no capítulo 4, quando explicamos como o computador armazena as variáveis criadas. Lembre-se que cada uma delas é alocada em uma posição na memória RAM e recebe um endereço. Assim, no exemplo apresentado, as variáveis estão em endereços diferentes e por isso a mudança em uma não afeta a outra (ilustrado na figura a seguir).

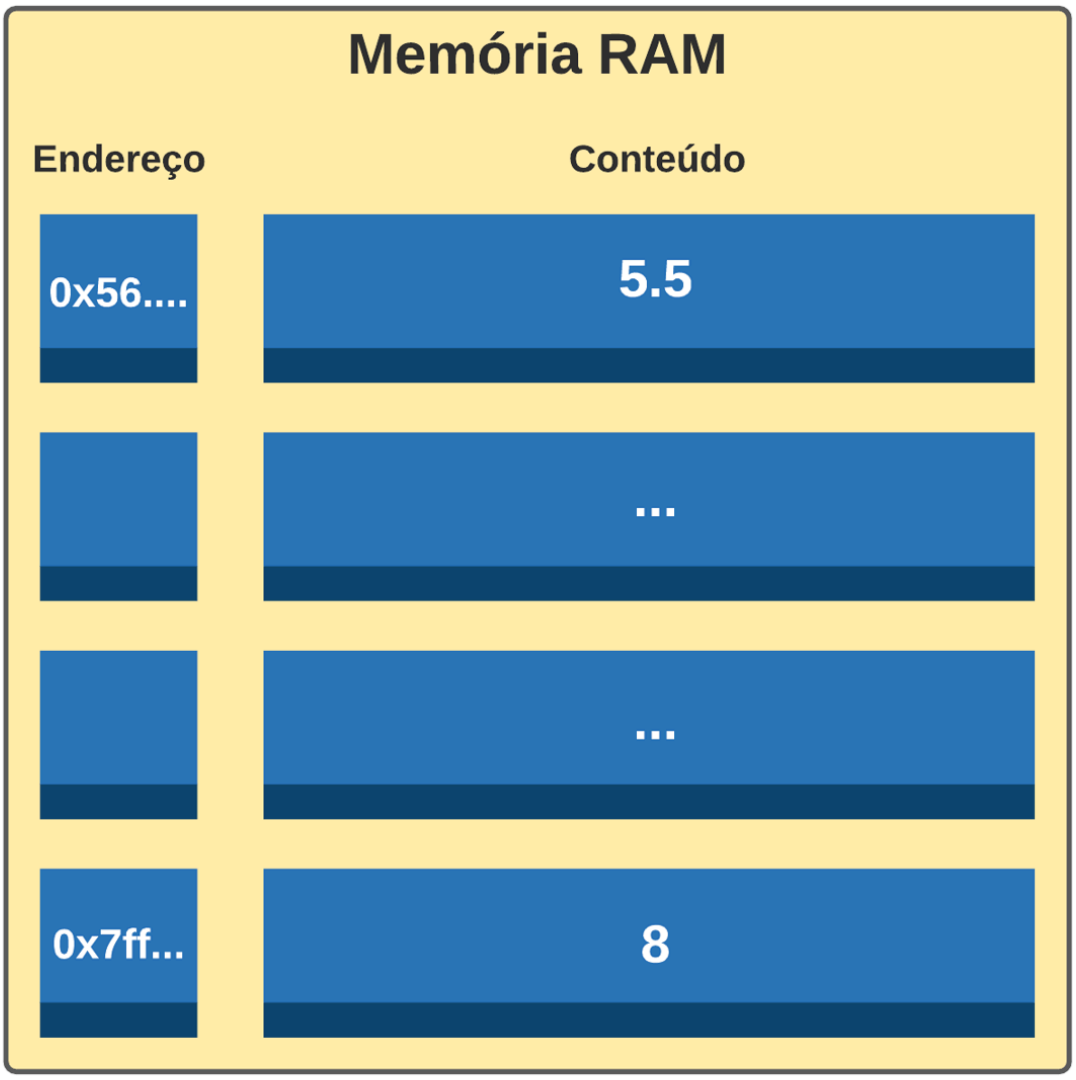

Figura 11.8: Representação das variáveis na memória do computador.

O conceito apresentado se aplica para variáveis do tipo numérica, *Strings* e booleanos. No entanto, quando os elementos de um dicionário são do tipo lista, tuplas, conjuntos, dicionários, entre outros, cria-se uma relação entre as variáveis, pois elas apontam para a mesma posição na memória. Vamos ilustrar essa situação com um dicionário que armazena uma lista. Na sequência, recuperamos essa lista e a guardamos em uma variável. Por fim,

modificamos o valor dessa variável para demonstrar que a lista original também foi afetada:

```
dicionario = dict(numeros = [ 1, 2, 3])
valor_guardado = dicionario["numeros"]
valor_guardado[0] = 5

print(dicionario["numeros"])
print(valor_guardado)
```

Na segunda linha, é criado um dicionário com a chave `numeros` e que guarda como valor uma lista. Na segunda linha recuperamos essa lista e a guardamos na variável `valor_guardado` . A alteração realizada com a instrução `valor_guardado[0] = 5` também modifica a lista que pertence ao dicionário, pois ambas apontam para a mesma posição na memória.

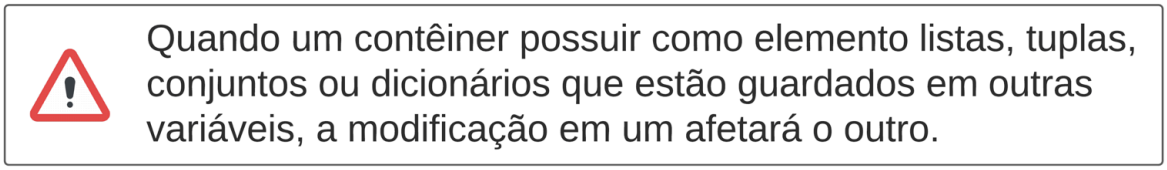

Figura 11.9: Quando um contêiner possuir como elemento listas, tuplas, conjuntos ou dicionários que estão guardados em outras variáveis, a modificação em um afetará o outro.

É importante entender o conceito de variáveis e endereços na memória e os impactos que essa relação entre as variáveis produz. Por vezes, você pode alterar uma variável sem saber que está modificando outra e o seu programa resultar em saídas inesperadas.

Variáveis que têm como valor outra variável do tipo String, numérico e booleano, não apresentam ligação entre si.

Figura 11.10: Variáveis que têm como valor outra variável do tipo String, numérico e booleano, não apresentam ligação entre si.

Inserção de elementos

A forma mais simples de inserir elementos em um dicionário é utilizar a sintaxe:

```
nome-do-dicionario[ chave ] = valor
```

Se a *chave* for uma *String* deve ser informada entre aspas. O código a seguir demonstra a aplicação da sintaxe apresentada:

```
notas = dict(LeonardoSoares = 5.5, GabrielFortes = 10, LinusTorvalds = 8,
SteveWozniak = 3)

notas["GuidoRossum"] = 10
print(notas)
```

O elemento de chave `GuidoRossum` é inserido no dicionário. Consequentemente, a impressão da lista final será formada pelos elementos: `{ 'LeonardoSoares' : 5.5, 'GabrielFortes' : 10, 'LinusTorvalds' : 8, 'SteveWozniak' : 3, 'GuidoRossum' : 10 }`.

Caso a sintaxe seja aplicada em uma chave já existente no dicionário, então o seu valor será atualizado:

```
notas = dict(LeonardoSoares = 5.5, GabrielFortes = 10, LinusTorvalds = 8,
SteveWozniak = 3)
```

```
notas["LeonardoSoares"] = 7
print(notas)
```

A saída do código será um dicionário formado pelos elementos: `{ ‘LeonardoSoares’ : 7, ‘GabrielFortes’ : 10, ‘LinusTorvalds’ : 8, ‘SteveWozniak’ : 3 }`

Remoção de elementos

Há diferentes formas de remover um elemento de um dicionário. A primeira é utilizar a função `pop` , que possui a seguinte sintaxe:

```
variavel = nome-do-dicionario.pop( chave )
```

Essa função remove um elemento do dicionário e após isso o retorna para uma variável. Também é importante notar que, caso a chave informada não exista, um erro do tipo `KeyError` será lançado pelo Python. A seguir, exemplifica-se o uso dessa função:

```
notas = dict(LeonardoSoares = 5.5, GabrielFortes = 10, LinusTorvalds = 8, SteveWozniak = 3)

nota_leonardo = notas.pop("LeonardoSoares")

print(notas)
```

Após a execução, o código apresentará a saída: `{ ‘‘GabrielFortes’ : 10, ‘LinusTorvalds’ : 8, ‘SteveWozniak’ : 3 }`

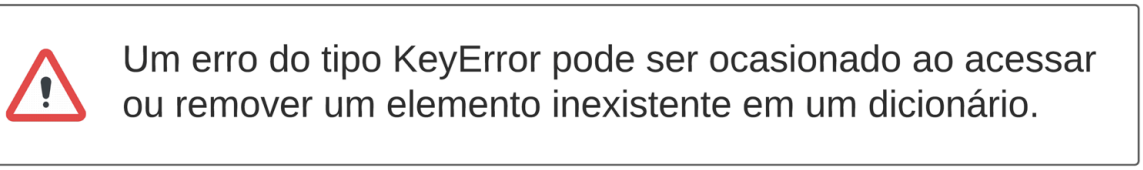

Figura 11.11: Um erro do tipo KeyError pode ser ocasionado ao acessar ou remover um elemento inexistente em um dicionário.

A instrução `del` também pode ser utilizada para remoção de elementos em um dicionário. Sua sintaxe é escrita da seguinte forma:

```
del nome-do-dicionario[ chave ]
```

```
notas = dict(LeonardoSoares = 5.5, GabrielFortes = 10, LinusTorvalds = 8,
SteveWozniak = 3)

del notas["LeonardoSoares"]

print(notas)
```

Esse código apresenta a mesma saída do anterior, onde foi utilizada a função `pop`. No entanto, `del` não retorna o elemento excluído.

Qual formato será escolhido depende do que será feito com o elemento removido. Caso precise desse dado para realizar algo, como apresentar uma mensagem na tela contendo essa informação, então é preciso utilizar a função `pop`. Caso contrário, as duas formas de remoção podem ser utilizadas.

Iteração pelos seus elementos

A instrução `for` é utilizada para iterar os elementos de um dicionário. A seguir apresentamos diferentes formas para realizar esse procedimento.

No primeiro formato, a iteração possibilitará o acesso às chaves do dicionário. Sua sintaxe é representada a seguir:

```
for chave_dicionario in nome-dicionario:
```

Em cada iteração do `for`, uma chave do dicionário é guardada na variável `chave_dicionario`. Com essa informação podemos acessar os dados do dicionário. Veja um exemplo de uso desse recurso:

```
notas = dict(LeonardoSoares = 5.5, GabrielFortes = 10, LinusTorvalds = 8,
SteveWozniak = 3)

for nome_estudante in notas:
    nota = notas [ nome_estudante ]
    print(nome_estudante)
    print(nota)
```

A saída do código apresentado será:

```
LeonardoSoares
5.5
GabrielFortes
10
LinusTorvalds
8
SteveWozniak
3
```

Outro formato de iteração faz uso da função `values`, que apresenta a sintaxe:

for valor in nome-do-dicionario.values():

Neste formato não possuímos acesso às chaves do dicionário, apenas o acesso direto aos seus valores. O exemplo anterior foi modificado para uso de `values` e resultará na mesma saída:

```
notas = dict(LeonardoSoares = 5.5, GabrielFortes = 10, LinusTorvalds = 8,
SteveWozniak = 3)

for nota in notas.values():
    print(nota)
```

Por fim, uma combinação das duas formas é obtida por meio da função `items`. Em cada iteração temos acesso a uma chave e ao seu respectivo valor. Sua sintaxe é apresentada a seguir:

for chave_dicionario , valor_dicionario in nome-dicionario:

O uso dessa função é representado no código:

```
notas = dict(LeonardoSoares = 5.5, GabrielFortes = 10, LinusTorvalds = 8,
SteveWozniak = 3)

for chave, nota in notas.items():
    print(chave)
    print(nota)
```

Resultando na saída:

```
LeonardoSoares
5.5
GabrielFortes
10
LinusTorvalds
8
SteveWozniak
3
```

A escolha pela forma de iteração dependerá das suas necessidades. Em alguns casos você precisa da chave para fazer alguma operação (por exemplo, definir um cálculo diferenciado para o aluno `LinusTorvalds`), em outros você precisa apenas dos valores. Assim, fica ao seu critério qual utilizar.

Operações em dicionários

As funções a seguir são próprias do Python e realizam ações em um dicionário.

- **clear**

Remove todos os elementos de um dicionário, transformando-o em um dicionário vazio:

```
notas = dict(LeonardoSoares = 5.5, GabrielFortes = 10, LinusTorvalds = 8,
SteveWozniak = 3)
notas.clear()
print(notas)
```

Saída: `{ }`

- **keys**

Retorna as chaves de um dicionário.

```
notas = dict(LeonardoSoares = 5.5, GabrielFortes = 10, LinusTorvalds = 8,
SteveWozniak = 3)
chaves = notas.keys()
print(chaves)
```

Saída:

```
 dict_keys(['LeonardoSoares', 'GabrielFortes', 'LinusTorvalds',
'SteveWozniak'])
```

- **copy**

Retorna uma cópia do dicionário. A cópia criada é independente da original e por isso a alteração em um dicionário não impactará o outro:

```
notas = dict(LeonardoSoares = 5.5, GabrielFortes = 10, LinusTorvalds = 8,
SteveWozniak = 3)

copia_notas = notas.copy()
```

Questões

1) Quais as partes que compõem um dicionário?

2) Qual a sintaxe para criação de um dicionário?

3) Crie um dicionário para armazenar o número de três amigos. A chave deve ser o nome e o sobrenome dessas pessoas.

4) Realize a iteração sobre o dicionário criado na questão anterior para imprimir o nome e sobrenome da pessoa seguido do seu número de contato.

Respostas

1) Quais as partes que compõem um dicionário?

R: Um dicionário é formado por valores que se associam às chaves.

2) Qual a sintaxe para criação de um dicionário?

R: Há diferentes formas para criar um dicionário, sendo uma delas por meio da sintaxe:

```
nome-dicionario = { chave : valor }
```

E o outro formato utiliza a função `dict` :

```
nome-dicionario = dict( chave = valor )
```

3) Crie um dicionário para armazenar o número de três amigos. A chave deve ser o nome e o sobrenome dessas pessoas.

```
agenda_telefonica = { "DonaldKnuth" : "351-853", "GraceHopper" : "876-453", "RobertoLerusalimschy": "518-987" }
```

4) Realize a iteração sobre o dicionário criado na questão anterior para imprimir o nome e sobrenome da pessoa seguido do seu número de contato.

```
agenda_telefonica = { "DonaldKnuth" : "351-853", "GraceHopper" : "876-453", "RobertoLerusalimschy": "518-987" }

for nome, telefone in agenda_telefonica.items():
    print(f"{nome} - {telefone}")
```

11.4 Matrizes

Ainda que os contêineres apresentados atendam a nossas necessidades na maioria dos casos, há situações que exigem formas mais complexas para representação dos dados.

Vamos relembrar o exemplo de notas, onde cada estudante possuía apenas uma nota. No entanto, é comum que em uma mesma disciplina os estudantes realizem mais de uma atividade avaliativa e consequentemente possuam mais de uma nota. Como representar essa situação?

No exemplo a seguir, vamos considerar o caso onde cada estudante realizou três avaliações. Para representá-las vamos utilizar listas:

```
notas_avaliacao_um = [ 10, 5.5, 7, 8, 6, 7, 9.5, 6.3, 6, 2.4 ]
notas_avaliacao_dois = [ 8, 6.5, 4, 9.2, 10, 3.1, 5, 8, 6, 7.4 ]
notas_avaliacao_tres = [ 6.5, 7, 6.2, 8.7, 8, 7.5, 9, 7.3, 9, 10 ]
```

No exemplo, temos três listas e cada uma armazena as notas dos mesmos alunos em diferentes avaliações. Apesar de o código atender as necessidades da situação apresentada, ele recai no mesmo problema do início do capítulo: **há variáveis que possuem relação entre si, mas estão representadas de forma separada**. Portanto, os contêineres apresentados não são satisfatórios para esse tipo de problema.

Quando falamos sobre qual problema os contêineres resolvem, explicamos que um deles é agrupar informações associadas em uma única variável, pois, além de organizar melhor o código, os contêineres possuem funções que ajudam a manipular esse conjunto de dados. Assim, o código apresentado precisa ser refatorado para refletir o que aprendemos neste capítulo.

Para agrupar as notas é preciso introduzir um conceito conhecido pelos nomes de **matrizes**, **listas bidimensionais** ou **listas aninhadas**. A proposta é criar uma lista que tem como elementos outras listas. A seguir, ilustramos esse caso, resolvendo o problema

apresentado anteriormente. Assim, teríamos uma variável nomeada `notas`, que é formada por outras três listas, onde cada uma representa um grupo de notas.

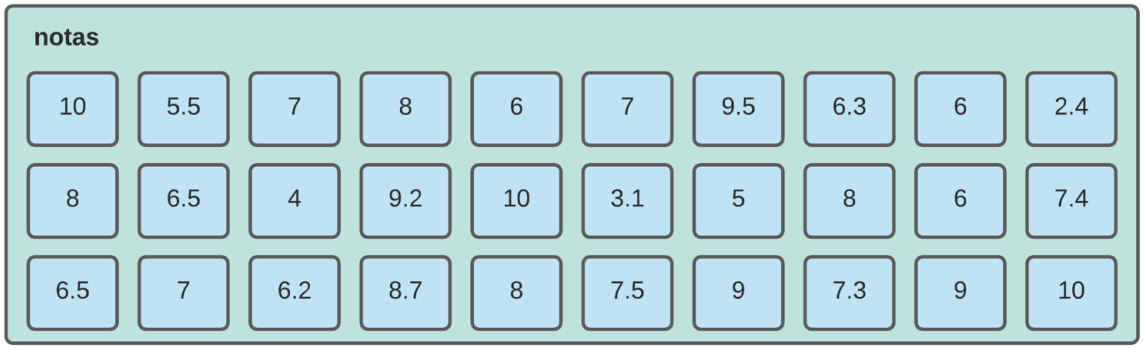

Figura 11.12: Representação de uma lista bidimensional.

Na programação, quando uma lista faz parte de outra, dizemos que há uma representação bidimensional. Para compreender isso é preciso entender que até o momento havíamos aprendido apenas sobre representações unidimensionais, onde cada elemento de uma lista está em uma determinada posição de 0 .. n-1.

Por outro lado, quando o elemento de uma lista é outra lista, isso significa que essa "sublista" também possui a sua própria dimensão de 0 .. n-1. Assim, há duas dimensões, uma da lista principal e outra dessa sublista, por isso o termo "bidimensional". Este conhecimento é importante, pois para acessar um elemento precisaremos saber das informações sobre as duas dimensões. É um pouco confuso, mas você já vivenciou esses conceitos na matemática quando estudou sobre matrizes. Na figura a seguir representamos uma matriz formada por 3 linhas e 4 colunas. Podemos dizer que a matriz equivale à lista principal e cada linha dessa matriz seria uma sublista.

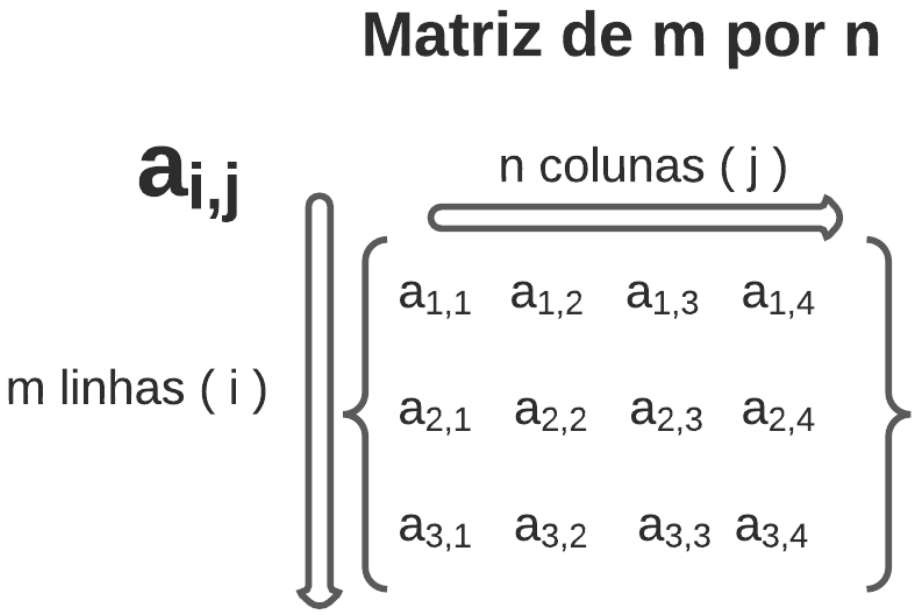

Figura 11.13: Representação de uma matriz matemática.

Como falamos anteriormente, o acesso a um elemento é feito a partir da sua posição que é formada por uma linha e coluna, sendo que a lista à qual o elemento pertence é indicado pela linha, e a sua posição nesta linha é indicado pelo número da coluna. Esse entendimento é fundamental, pois o Python e outras linguagens de programação seguem esse formato.

Uma matriz bidimensional é a representação de uma lista em que seus elementos são outras listas unidimensionais.

Figura 11.14: Uma matriz bidimensional é a representação de uma lista em que seus elementos são outras listas unidimensionais.

Voltando ao exemplo anterior das notas, poderíamos então organizá-los da seguinte forma:

	Col 1	Col 2	Col 3	Col 4	Col 5	Col 6	Col 7	Col 8	Col 9	Col 10
Linha 1	10	5.5	7	8	6	7	9.5	6.3	6	2.4
Linha 2	8	6.5	4	9.2	10	3.1	5	8	6	7.4
Linha 3	6.5	7	6.2	8.7	8	7.5	9	7.3	9	10

Figura 11.15: Representação bidimensional das notas.

Para fins de análise, considere que cada coluna representa um aluno e cada linha representa sua nota em uma determinada atividade avaliativa. Portanto, a nota localizada na linha 1 e coluna 1 pertence ao primeiro aluno na primeira avaliação; e a nota localizada na linha 2 e coluna 2, pertence ao primeiro aluno na segunda avaliação. Seguindo esse raciocínio, a nota na linha 1 e coluna 5 pertence ao quinto aluno na primeira avaliação.

Sabendo destes conceitos podemos agora representá-los em um código de programação. A primeira etapa é realizar a criação da lista bidimensional. Sua sintaxe é representada pela instrução:

```
nome-da-matriz = [ [ elementos-da-lista ], [ elementos-da-lista ] ]
```

Perceba que cada elemento da lista é formado por um par de colchetes. Dentro deles devemos informar os seus elementos, seguindo o formato que já aprendemos. O código a seguir apresenta o uso dessa sintaxe:

```
notas = [
[10, 5.5, 7, 8, 6, 7, 9.5, 6.3, 6, 2.4],
[ 8, 6.5, 4, 9.2, 10, 3.1, 5, 8, 6, 7.4 ],
```

```
[ 6.5, 7, 6.2, 8.7, 8, 7.5, 9, 7.3, 9, 10 ]
]
```

Observe a sintaxe, onde há uma variável `notas` , que é uma lista e cada um dos seus elementos, três no total, são outras listas. Este exemplo demonstra a criação de uma matriz 3 x 10, ou seja, formada por três linhas (listas) e dez colunas (número de elementos em cada lista).

Acesso e modificação dos elementos

O acesso aos elementos de uma matriz pode ser realizado de duas formas. Inicialmente podemos acessar cada sublista individualmente com a sintaxe:

```
lista = nome-da-matriz[ índice ]
```

Desta forma, a variável `lista` será uma das sublistas identificadas pelo índice informado. O acesso aos seus elementos segue o formato já conhecido:

```
elemento = lista[ índice ]
```

Por exemplo, no código a seguir acessamos a primeira lista de notas e na sequência a quinta nota desta lista:

```
notas = [
[10, 5.5, 7, 8, 6, 7, 9.5, 6.3, 6, 2.4],
[ 8, 6.5, 4, 9.2, 10, 3.1, 5, 8, 6, 7.4 ],
[ 6.5, 7, 6.2, 8.7, 8, 7.5, 9, 7.3, 9, 10 ]
]

primeira_avaliacao = notas[0]
quinta_nota = primeira_avaliacao[4]

print(quinta_nota)
```

Resultará na saída: `6`

Podemos ainda acessar um elemento da sublista sem precisar guardá-la em uma variável. Para isso, deveremos informar o índice da sublista, seguido do índice do elemento que desejamos, seguindo a sintaxe:

```
elemento = nome-da-matriz[ índice_sublista][ índice ]
```

O código anterior foi refatorado para exemplificar essa nova sintaxe e produz o mesmo resultado:

```
notas = [
[10, 5.5, 7, 8, 6, 7, 9.5, 6.3, 6, 2.4],
[ 8, 6.5, 4, 9.2, 10, 3.1, 5, 8, 6, 7.4 ],
[ 6.5, 7, 6.2, 8.7, 8, 7.5, 9, 7.3, 9, 10 ]
]

quinta_nota = notas[0][4]

print(quinta_nota)
```

Sugerimos a leitura desta seção mais de uma vez, pois os conceitos são complexos. É muito importante compreendê-los, pois listas bidimensionais possuem inúmeras aplicações na computação. Por exemplo, um tabuleiro de xadrez pode ser representado com esse tipo de contêiner, onde cada uma de suas casas seria acessada por uma linha e coluna respectivamente.

Por fim, é ainda possível ter listas de três ou mais dimensões, bastando seguir o formato onde uma lista é formada por outra e por outra e assim em diante:

```
lista = [ [ [ 1, 2, 3], ["a", "b", "c"] ], [ [ "d", "e", "f" ] ] ]
elemento_sub_lista_1_item_2_2 =  lista[0][1][1]
```

A execução resultará na saída `b` . É um código confuso, mas serve para ilustra as possibilidades de uso deste recurso.

Iteração pelos elementos

A iteração sobre uma matriz apresenta similaridades com as listas convencionais. Faremos uso do `for`, mas agora vamos utilizar dois, um para iterar pela matriz e outra para iterar pelos elementos da sublista.

No código a seguir, apresenta-se a nota dos estudantes em cada uma das avaliações:

```
notas = [
[10, 5.5, 7, 8, 6, 7, 9.5, 6.3, 6, 2.4],
[ 8, 6.5, 4, 9.2, 10, 3.1, 5, 8, 6, 7.4 ],
[ 6.5, 7, 6.2, 8.7, 8, 7.5, 9, 7.3, 9, 10 ]
]

for m in range(len(notas)):
    notas_avaliacao = notas[m]
    for n in range(len(notas_avaliacao)):
        print(f"Nota na avaliação {m+1}: {notas[m][n]}")
```

Observe que a primeira instrução do primeiro `for` é o acesso a uma das sublistas, procedimento feito pela instrução `notas[m]`, onde a variável *m* é apenas um contador fornecido por `range` na iteração da matriz. De posse da sublista, podemos iterá-la com outro `for`.

Vamos recapitular como o Python trabalha em situações como a apresentada onde há repetições aninhadas. Na figura a seguir, ilustramos o cenário da primeira iteração do loop principal e as demais do loop secundário.

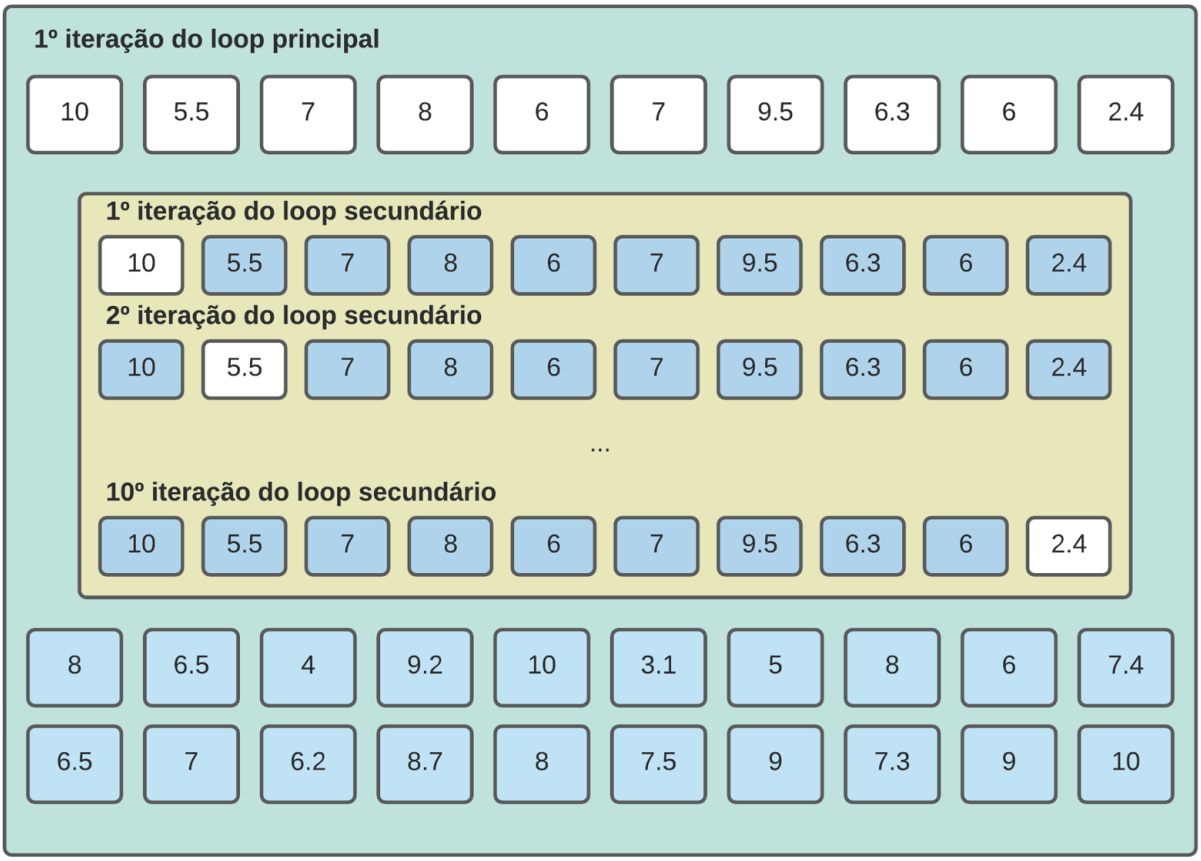

Figura 11.16: Representação bidimensional das notas.

Em cada iteração da lista principal temos acesso a uma de suas sublistas. Quando se inicia a iteração do loop secundário, cada elemento da sublista será acessado. Após o término das dez iterações do loop secundário, o loop principal passará para a segunda iteração e repetirá os procedimentos descritos anteriormente.

Uma repetição aninhada pode acessar as variáveis declaradas na repetição principal.

Figura 11.17: Uma repetição aninhada pode acessar as variáveis declaradas na repetição principal.

11.5 Erros comuns

Não tratar a tentativa de deleção de elementos inexistentes em um conjunto. Uma tentativa de deleção de um elemento inexistente de um conjunto provocará um erro do tipo `KeyError`. Assim, é fundamental garantir que essa ação será feita em um item existente:

Código com problema:

```
dados = { 354, 456 }
dados.remove( 789 )
```

Código corrigido:

```
dados = { 354, 456 }
if 789 in dados:
    dados.remove( 789 )
```

Não informar as aspas para as chaves durante a criação de um dicionário. A sintaxe de criação de um dicionário formado por chaves *String* deve utilizar aspas, caso seja realizada com { }.

Código com problema:

```
dados = { ano: 1987 }
```

Código corrigido:

```
dados = { "ano": 1987 }
```

Questões

1) Qual a sintaxe para criação de uma matriz de duas dimensões?

2) Como se dá o acesso aos elementos de uma matriz?

3) Crie uma matriz numérica de 3x3, com o primeiro elemento iniciando em 10 e finalizando em 18.

4) Considerando o código anterior, escreva a sintaxe para acessar o elemento na posição `[1][2]` e indique o seu valor.

Respostas

1) Qual a sintaxe para criação de uma matriz de duas dimensões?

nome-da-matriz = [[elementos], [elementos], [elementos]]

2) Como se dá o acesso aos elementos de uma matriz?

R: Por meio do índice linha e do índice coluna. O primeiro refere-se ao índice da lista principal, enquanto o segundo é um índice da lista secundária.

3) Crie uma matriz numérica de 3x3, com o primeiro elemento iniciando em 10 e finalizando em 18.

```
matriz = [ [10, 11, 12], [13, 14, 15], [16, 17, 18] ]
```

4) Escreva a sintaxe para acessar o elemento na posição `[1][2]` e indique o seu valor.

```
matriz = [ [10, 11, 12], [13, 14, 15], [16, 17, 18] ]

print(matriz[1][2])
```

R: O elemento possui valor 15.

11.6 Resumo e mapa mental

- Neste capítulo aprendemos sobre novos tipos de contêineres: conjuntos, dicionários e matrizes;
- Os conjuntos são uma representação de dados que é útil para situações em que se deseja verificar a existência de elementos

duplicados, diferenças e intersecções entre conjuntos. Além disso, possui uma restrição de não permitir dados duplicados, o que é interessante para casos onde isso é necessário;
- Os dicionários representam seus elementos por meio de chaves e valores. Oferece maior semântica aos dados em comparação às listas e tuplas com seus índices numéricos;
- Matrizes possibilitam a representação de estruturas de dados mais complexas formadas por múltiplas listas.

Não permite elementos duplicados
Itersecção
União
Diferença
Iteração
Acesso aos elementos por
Operações
Conjuntos
Representação bidimensional de uma lista
Contêineres
Matrizes
Dicionários
Operações
Representação
Consulta
Deleção
Inserção
Iteração
Chave
Valor
Valor
Chave
Chave e Valor

Figura 11.18: Mapa mental do capítulo 11.

CAPÍTULO 12
Tópicos intermediários em programação

1. O que você já sabe?

Os conteúdos abordados até o momento (variáveis, condições, repetições, funções e contêineres de dados) são a base da programação introdutória e presentes na maioria dos softwares. Se você acompanhou o livro do começo até este capítulo, então já está pronta(o) para explorar conceitos mais avançados que são igualmente importantes.

2. O que veremos?

Neste capítulo apresentaremos temas intermediários da programação que são necessários mesmo para iniciantes na área. Além de novos conceitos, também vamos nos aprofundar em assuntos anteriores, estabelecendo uma importante base para a sua carreira em programação. Para finalizar, vamos sugerir um caminho de estudos a partir de onde você se encontra, que será útil para dar os próximos passos necessários para se tornar um programador ou programadora de excelência.

3. Por que é importante?

Um software é construído a partir de múltiplos conceitos da programação. A base apresentada neste livro é fundamental para iniciar na área, mas existem outros que são igualmente importantes.

Muitos dos recursos apresentados neste capítulo não existiam nas versões iniciais do Python e foram introduzidos com o objetivo de melhorar o processo de desenvolvimento de software. Conhecê-los é importante, pois facilitará o desenvolvimento de programas, seguindo as melhores práticas de desenvolvimento dessa linguagem de programação.

O que você vai aprender?

Ao término deste capítulo você compreenderá:

- Como realizar o tratamento de erros no Python;
- Tópicos avançados em variáveis;
- Criação das estruturas de dados do tipo pilha e filas;
- A aplicação das instruções de yield e generators;
- Tópicos avançados em funções.

12.1 Tópicos avançados em variáveis

O primeiro conteúdo de programação abordado neste livro apresentou o conceito de variáveis. Elas representam a base de toda a programação, sendo utilizadas por todos os demais recursos do Python. Vamos retomar este tema, aprofundando os conhecimentos sobre os tipos de variáveis existentes no Python.

Tipos de variáveis

Em algumas linguagens de programação, como o Java, na criação de uma variável é preciso escrever o seu tipo. Por exemplo, a criação de uma *String* e um inteiro nesta linguagem são feitas com a seguinte instrução:

```
String pais = "Brasil";
int numero = 2;
```

Observe que antes dos nomes das variáveis é preciso escrever os seus respectivos tipos. Linguagens que seguem esse formato são chamadas de **fortemente tipadas**, pois sem essa definição violaríamos sua sintaxe. O Python, por outro lado, é uma linguagem **fracamente tipada** e por isso não requer que na declaração de uma variável o seu tipo seja definido:

```
pais = "Brasil"
numero = 2
```

Há vantagens e desvantagens das linguagens fracamente e fortemente tipadas e não compete ao escopo deste livro discuti-las. Nesta seção vamos explorar os tipos existentes no Python. Ainda que implícitos, eles são considerados internamente pelo interpretador e é importante conhecê-los para saber como a linguagem trata os diferentes dados que você utiliza em seu programa.

Para conhecer os tipos das variáveis pode-se utilizar a função `type`, que apresenta a sintaxe:

```
tipo-variavel = type( variavel )
```

Observe que a função recebe como parâmetro uma variável e retornará o tipo que ela possui. Seu uso é representado no código a seguir:

```
pais = "Brasil"
numero = 2
tipo_pais = type(pais)
tipo_numero = type(numero)
print(tipo_pais)
print(tipo_numero)
```

O código apresentará como saída:

```
<class 'str'>
<class 'int'>
```

O Python utiliza essa representação para indicar os tipos *String* e número inteiro, respectivamente.

Existe uma variedade de tipos no Python. A tabela a seguir apresenta os principais:

Categoria	Tipos
Números	int, float e complex
Texto	str
Sequências	list, tuple e range
Mapeamento	dict
Conjunto	set e frozenset
Booleanos	bool
Binários	bytes, bytearray e memoryview

Pela tabela, observam-se alguns tipos de variáveis não abordados até então, como *complex*, *bytes*, *bytearray* e *memoryview*.

O tipo *complex* é uma representação para números complexos, aqueles que você estudou na escola. Talvez não saiba onde você vai utilizá-los, mas se a sua aplicação envolve algo de trigonometria, física e engenharia aeronáutica, entre outras, há uma chance de você precisar deste tipo de dado. A sintaxe para uso deste tipo de dado é representada a seguir:

```
variavel = x + yj
```

Onde x e y são números reais e *j*, a parte imaginária. No Python um número complexo pode ser representado da seguinte forma:

```
z = 3+4j
print( type(z) )
```

A variável *z* armazena um número complexo e o `print` utilizado vai apresentar a saída `<class 'complex'>`. Pela fórmula apresentada, os números *3* e *4* são as partes reais, enquanto *j* é a parte imaginária.

O Python é uma linguagem fracamente tipada. Ou seja, não é preciso definir o tipo que a variável possui durante a sua criação.

Figura 12.1: O Python é uma linguagem fracamente tipada. Ou seja, durante a criação de uma variável não é preciso escrever o seu tipo.

Um outro tipo de dado que pode ser armazenado no Python são os *bytes*. Para quem é iniciante na computação pode estranhá-lo, mas juntamente com os bits, eles formam a base da nossa computação atual. Tudo o que computadores, smartphones, entre outros dispositivos eletrônicos armazenam e processam são chamados de *bits*. Um *bit* (dígito binário) é uma informação representada por um valor que pode ser 0 ou 1 (por isso recebe o nome de dado binário). Quando temos um conjunto de 8 bits forma-se um byte, como ilustrado na figura a seguir.

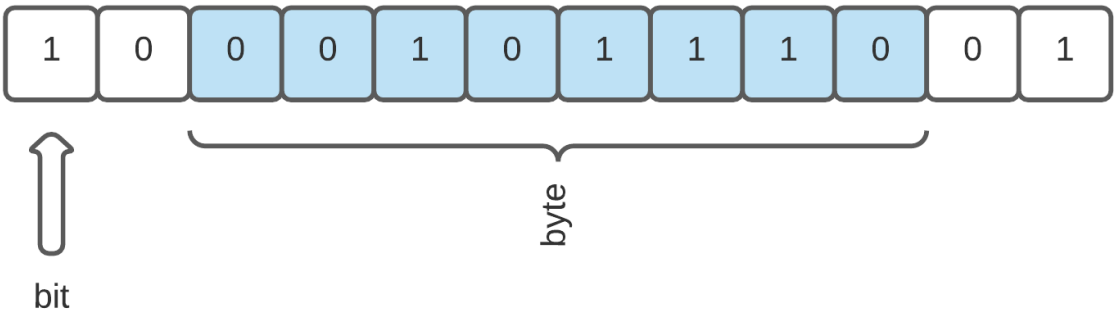

Figura 12.2: Representação de bits e bytes.

Se você está lendo este livro em um leitor de PDFs, para o computador ele é um conjunto de bytes. Uma imagem publicada em sua rede social preferida, o texto do seu e-mail, um arquivo de música, todos esses dados são representados por meio de bits e bytes. Ou seja, na computação, tudo se resume a um conjunto de vários 0s e 1s. Isso é um pouco estranho no começo, mas é a forma como o computador representa dados em seus sistemas eletrônicos.

Gostaria de fazer um teste? Crie um arquivo de textos no editor de textos de sua preferência e escreva a seguinte frase com 23 caracteres: "Estou amando este livro" (sem as aspas) e salve-o com uma extensão .txt. Ao abrir as propriedades deste arquivo é possível observar que ele possui 23 bytes de tamanho. Isso ocorre, pois o computador utiliza um conjunto de 1 byte (ou 8 bits, como preferir) para cada caractere em um texto. Letras com acentos e outros caracteres especiais demandam mais bytes de armazenamento.

Na figura a seguir, apresenta-se a representação do texto anterior nos formatos binário e hexadecimal. Este segundo formato também é bastante utilizado na computação e consiste de um sistema de numeração formado por 16 algarismos. Dados hexadecimal existem, pois são uma forma mais amigável de representar informações. Assim, em muitos casos os números binários são representados em hexadecimais. Voltando para a imagem, destacamos duas letras, o 'E' em maiúsculo e a letra 'o' minúscula. Como dito anteriormente, cada letra ocupa 8 bits que formam uma combinação de 0s e 1s única para cada uma. Como são duas letras diferentes, consequentemente a representação binária deles também será.

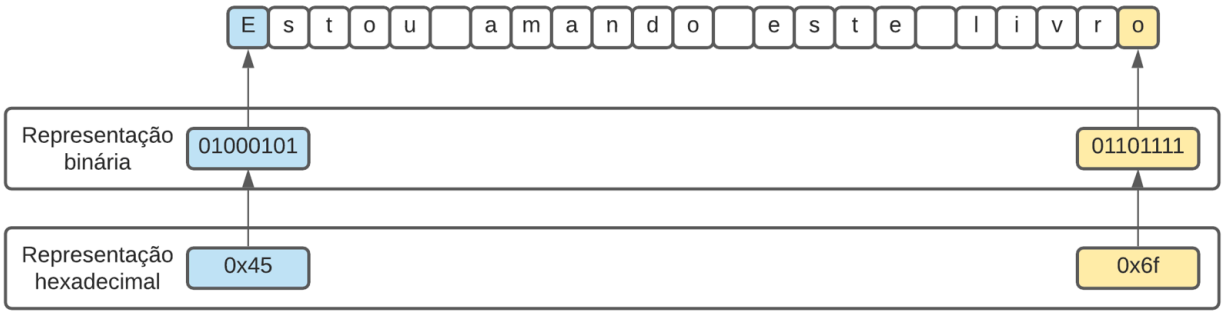

Figura 12.3: Representação de um texto em bits e bytes.

Também podemos confirmar que os computadores guardam as informações em bits e bytes ao abrir um arquivo de imagem em um editor de textos. Você visualizará algo parecido com o trecho a seguir:

```
\89PNG
\00\00\00
IHDR\00\003\00\009\00\00\00\EC[U\A1\00\00\00gAMA\00\00\B1\8F
\FCa\00\00\00
cHRM\00\00z&\00\00\80\84\00\00\FA\00\00\00\80\E8\00\00u0\00\00\EA`\00\00:\
98\00\00p\9C\BAQ<\00\00:PLTE\FF\FF\FF\F6\F6\F6\E5\E5\E5\EE\EE\B7333zzz\CF\
CF\9A\9A\DB\DBჯ\B\8BWWW\FB\FB\FBFFFiii\A8\A8\A8\C3\C3\A6\A6\EA\EA\EA\D0\D
0Я\AF\AF\88\88\88]]]\F8\F8\F8\E2\E2\E2\C6\C6l\96\96mmm\FC\FC\FC\BA\BA\BA\C
4\C4\C4\F1\F1\F1p\8D\A2x\A0\BD~\AF\BB\E5z\8E\E2ĸ\F8\E5\A5\FF\A3\A3dnv\BA\A
F\89\FC\E9\A7m\86\98\82\B9\E2\EAafi|\AA̤\83rgv\82v\9D\B8mjdy\A4Ṛ\8Fx\D0\A3\
9B~\F1\E0\A2\F9
```

Esse conjunto estranho de caracteres formam a representação binária da imagem, ou seja, a forma como o dado é armazenado no computador. Apesar de serem binários, os dados são apresentados para você no formato hexadecimal, pois o editor de textos realiza essa conversão.

Assim, aquela bela imagem em seu computador é armazenada internamente como 0s e 1s. Para que você possa visualizá-la, o computador realiza o processamento desses dados binários em pixels que serão exibidos no monitor.

Felizmente, para nós programadores(as), na maioria dos casos não precisamos lidar com os dados binários diretamente, pois linguagens como o Python abstraem certos elementos e facilitam nossa vida. Ainda assim, há casos em que pode ser preciso manipular bytes diretamente, e algumas funções são disponibilizadas.

A representação de bytes pode ser feita por meio das funções `bytes`, `bytearray` e `memoryview`. A primeira resulta em uma variável que representa bytes, mas é imutável, enquanto a segunda possibilita que seus elementos sejam modificados. A seguir criamos uma variável do tipo `bytes` que contém o caractere `E`, lembrando que a sua representação hexadecimal é `0x45`. No Python representamos esse dado como `\x45`:

```
letra_bytes = bytes(b"\x45")
```

Podemos ainda converter o byte anterior em uma forma mais amigável, como uma *String*, por meio da função `decode`:

```
letra_bytes = bytes(b"\x45")
letra_string = letra_bytes.decode()
print(letra_string)
```

Assim, enquanto a variável `letra_bytes` armazena uma representação em bytes, a variável `letra_string` armazena a conversão destes bytes em uma *String* legível para nós, humanos. A saída desse código será a impressão da letra `E`.

Por fim, variáveis do tipo *memoryview* possibilitam a manipulação de bytes de forma eficiente, o que é útil quando se trabalha com grandes dados. Por exemplo, a depender da ação realizada nos dados binários, o Python realiza uma cópia deles na memória, que em muitos casos é desnecessária. O uso de `memoryview` evita essa situação. Não abordaremos a sintaxe e uso dessas funções, pois foge ao escopo de programação introdutória. Essas informações podem ser obtidas na documentação oficial do Python (https://docs.python.org/3/library/stdtypes.html#bytes).

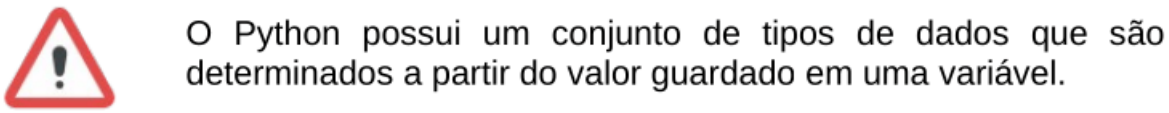

Figura 12.4: O Python define o tipo de uma variável a partir do valor armazenado por ela.

12.2 Tópicos avançados em funções

Aprendemos que as funções possuem um nome, parâmetros, escopo e retorno. Nesta seção vamos expandir o conhecimento sobre parâmetros e retorno, apresentando novos formatos de uso para eles. Também vamos apresentar o conceito de `generators`, que são úteis para funções que trabalham com um grande volume de dados.

Quantidade variável de parâmetros

Para relembrar, os parâmetros possibilitam o envio de dados para uma função que pode receber zero ou mais parâmetros. Esse quantitativo deve ser previamente definido no momento da criação da função. Para executá-la, é preciso saber sobre quantos parâmetros são necessários. Por exemplo, a seguir há uma função que calcula a soma de dois números, recebendo-os como parâmetros. Assim, na sua execução devemos informar quais serão esses dados.

```
def somar(numero_um, numero_dois):
    resultado_soma = numero_um + numero_dois
    return resultado_soma

resultado = somar(2, 2)
print(resultado)
```

Ainda que a função apresentada consiga calcular corretamente uma soma, ela está limitada ao uso de dois números por vez. Para aumentar esse quantitativo seria preciso incluir outros parâmetros na declaração da função. Ainda assim, sempre estaríamos limitados à quantidade informada.

Para suprimir a limitação apresentada, podemos definir que nossa função receberá um número variável de parâmetros. Não importa se serão dois, nove ou cem parâmetros, todos os casos serão aceitos. Realizar isso vai de encontro com o que aprendemos, pois era preciso definir quantos parâmetros nossa função receberia. Nesta seção vamos aprender como contornar essa restrição.

A definição do número variável de parâmetros é feita por meio do operador de desempacotamento (do inglês *unpacking operator*): `*`. Utilizamos esse operador durante a declaração de uma função seguindo a sintaxe:

```
nome-da-funcao( *dados )
```

Ao utilizar esse operador o Python permitirá que na execução da função sejam informados múltiplos parâmetros que serão "armazenados" na variável `dados`, que será do tipo tupla. Vamos ilustrar esse conceito no código a seguir:

```
def somar( *numeros ):
    resultado_soma = 0
    for numero in numeros:
        resultado_soma += numero
    return resultado_soma
resultado = somar(2, 2, 4, 6, 8)
print(resultado)
```

Com a execução do código a variável `resultado` armazenará o valor 22. Observe na primeira linha, onde está a declaração da função `somar`, que ela recebe como parâmetro uma única informação: `*numeros`. Isso indica ao Python que a nossa função recebe zero ou mais parâmetros. Na linha 6 é feita a execução desta função, perceba como múltiplos parâmetros são informados.

Todos os números informados serão armazenados na variável `numeros`. Por essa razão, foi possível iterar em seus dados para recuperar cada um dos parâmetros informados (linha 3).

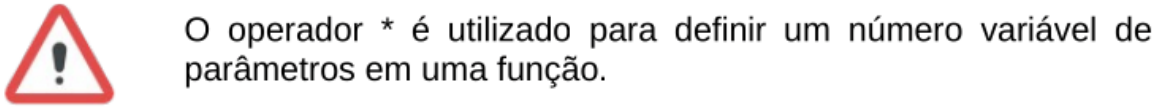

Figura 12.5: O operador * é utilizado para definir um número variável de parâmetros em uma função.

Também é possível definir um número variável de parâmetros que serão informados por meio de *keywords*. Neste formato, indica-se explicitamente para qual parâmetro um determinado dado está sendo enviado. Para relembrar, no capítulo 9 apresentamos o seguinte exemplo:

```
def dizer_ola(nome_usuario, sobrenome):
    print("Olá", nome_usuario, sobrenome)
```

```
dizer_ola(sobrenome = "Fortes", nome_usuario = "Pietra")
```

Observe que na chamada da função a ordem dos parâmetros está invertida em comparação à forma como foram declarados, com o sobrenome sendo informado primeiro. Mesmo assim, os dados foram enviados aos parâmetros adequados, pois seus nomes foram informados explicitamente.

Voltando ao nosso tema, para usar um número variável de parâmetros por *keyword* utiliza-se o operador `**` :

nome-da-funcao(**dados)

A variável `dados` será do tipo dicionário, onde as chaves serão os *keywords* e os valores o dado informado. Exemplificamos o uso dessa instrução com uma função que apresenta os dados de um veículo:

```
def exibir_informacoes_veiculo( **dados ):
    for dado, valor in dados.items():
        print(f"Informação {dado}, Descrição: {valor}")
exibir_informacoes_veiculo(ano = "1987", motor = "200cv")
```

A saída desse código será:

```
Informação ano, Descrição: 1987
Informação motor, Descrição: 200cv
```

Observe que o operador `**` foi utilizado na declaração da função (linha 1). A variável `dados` será iterada para acessar os dados passados como parâmetro. Como ela é do tipo dicionário, vamos utilizar a função `items` que nos permite acessar suas chaves e valores.

O operador ** é utilizado para definir um número variável de parâmetros por *keyword* em uma função.

Figura 12.6: O operador ** é utilizado para definir um número variável de parâmetros por keyword em uma função.

Os diferentes tipos de parâmetros que o Python oferece (tradicional, variável e variável por keyword) podem ser utilizados simultaneamente em uma única função. No entanto, na declaração da função é preciso seguir uma ordem para os parâmetros. Deve-se iniciar pelos parâmetros padrão, seguido das variáveis sem keyword (operador `*`) e variáveis com keyword (operador `**`).

A função a seguir apresenta os três tipos de parâmetros e a ordem em que devem ser utilizados em uma função:

```
def exibir_informacoes_veiculo(quantidade_veiculos, *nome, **dados):
    print(f"Total de veículos {quantidade_veiculos}")
    nome_usuario = ""
    for parte_nome in nome:
        nome_usuario += " "+parte_nome
    print(f"Nome completo: {nome_usuario}")
    for dado, valor in dados.items():
        print(f"Informação: {dado}, Descrição: {valor}")

exibir_informacoes_veiculo(1, "Gabriel", "Fortes", "de", "Macêdo", ano =
"1987", motor = "200cv")
```

O Python utiliza a ordem dos tipos de parâmetros para saber a quem atribuir os dados informados na execução da função. Observe a última linha do código onde a função é chamada, o primeiro parâmetro (inteiro 1) é atribuído à variável `quantidade_veiculos`, enquanto as *Strings* (Gabriel, Fortes, de e Macêdo) são atribuídas à variável `nome`. Por fim, os dados com keywords são atribuídos à variável `dados`. Uma mudança na ordem desses parâmetros ocasionará um erro `SyntaxError: invalid syntax`.

Retorno de múltiplos dados

As funções apresentadas até o momento retornavam apenas um dado. Quando apresentamos esse tema também comentamos sobre isso e destacamos essa restrição. De fato, isso ocorre com a maioria das linguagens de programação e muitos aprendizes acabam por cometer erros tentando retornar múltiplos dados. No entanto, o Python apresenta a possibilidade de retornar mais de uma informação.

Primeiramente vamos relembrar o conceito de retorno que possibilita à função devolver o resultado do seu processamento, que seria perdido no encerramento do seu escopo. Exemplificamos esse recurso com a função de soma a seguir.

```
def somar(numero_um, numero_dois):
    resultado_soma = numero_um + numero_dois
    return resultado_soma

resultado = somar(2, 2)
```

Apresentamos esse exemplo anteriormente e o replicamos aqui para relembrar sobre o conceito de retorno. Observe que dentro do escopo da função há uma variável `resultado_soma` e que utilizamos a instrução `return` para que o dado dessa variável possa ser guardado fora da função. Isso é observado na linha em que a função é chamada, onde seu retorno é guardado na variável `resultado`.

Para que a sua função retorne múltiplos dados devemos seguir a sintaxe:

```
return dado_um, dado_dois, ..., dado_n
```

Observe que os dados a serem retornados são separados por vírgula.

Para utilizar funções que retornam mais de uma informação devemos mudar a sintaxe de execução, como ilustrado pela sintaxe:

```
variavel_um, variavel_dois, ..., variavel_n = nome-da-funcao()
```

Observe que no momento de execução da função devemos indicar as variáveis que receberão o retorno. O quantitativo informado deve ser igual ao número de dados retornados. Exemplificamos esses conceitos na função a seguir que calcula a soma e subtração de dois números, retornando os resultados nessa ordem.

```
def calcular_soma_subtracao( numero_um, numero_dois ):
    resultado_soma = numero_um + numero_dois
    resultado_subtracao = numero_um - numero_dois
    return resultado_soma, resultado_subtracao
soma, subtracao = calcular_soma_subtracao(2, 2)
print(soma)
print(subtracao)
```

Com a execução do código, a variável `soma` armazenará o valor 4 e `subtracao` terá o valor 0.

Também é possível informar apenas uma variável para guardar o retorno e neste caso seu tipo será uma tupla.

```
def calcular_soma_subtracao( numero_um, numero_dois ):
    resultado_soma = numero_um + numero_dois
    resultado_subtracao = numero_um - numero_dois
    return resultado_soma, resultado_subtracao
resultado = calcular_soma_subtracao(2, 2)
print(resultado)
```

A saída para esse código será uma tupla `(4, 0)`. Relembrando este conceito, poderíamos acessar os elementos por meio dos seus índices `resultado[0]` e `resultado[1]`.

Atenção, pois o Python exige que os dados retornados sejam guardados em no mínimo uma variável ou em um número de

variáveis igual ao quantitativo de dados retornados. Por exemplo, em uma função que retorna três dados, pode-se executar a função com apenas uma variável ou três. Caso informe apenas duas, um erro do tipo `ValueError: too many values to unpack (expected 2)` será lançado.

Generators

O conceito de `generator` é um dos mais importantes quando pensamos sobre funções. Os iniciantes em programação dificilmente o utilizarão, porém à medida que os programas se tornam mais complexos esse recurso passa a ser necessário. Ele é especialmente útil para funções que realizam grande processamento de dados, pois há um elevado uso da memória do computador e que pode ocasionar problemas. Vamos exemplificar com um código que gera uma sequência de 10 milhões de números e os armazena uma lista:

```
def gerar_numeros():
    numeros = []
    for numero in range(10000000):
        numeros.append(numero)
    return numeros

numeros_gerados = gerar_numeros()
```

Sabemos que as variáveis de um programa são armazenadas na memória RAM do computador. Também aprendemos que um código é executado linearmente, de cima para baixo. Assim, o procedimento acima é relativamente custoso para a máquina, pois à medida que a iteração é executada, os dados são guardados na lista. É possível que após um tempo de execução o seu computador trave, pois ele está gerenciando milhões de dados na memória RAM, que crescem à cada iteração do `for`. Ao término da execução da função, a variável `numeros_gerados` será uma lista com 10 milhões de elementos.

Pensando em situações como a apresentada, a equipe do Python introduziu o recurso conhecido por `function generator`. A proposta evita que uma função realize todo o seu processamento de uma

única vez. A ideia é disponibilizar os dados processados sob demanda, evitando guardá-los na memória de uma única vez, como exemplificado no código anterior.

Segundo a proposição do `function generator`, se a função anterior fizesse uso deste recurso, o "retorno" dos dados seria feito de forma gradual, um por vez, totalizando 10 milhões de dados retornados. Colocamos entre aspas, pois o que ocorre não é o retorno produzido pela instrução `return`, mas a entrega de um dado da função para fora dela, que seria o equivalente do ponto de vista da devolução da informação. No entanto, isso difere do retorno, pois este encerra a execução da função, enquanto que no `function generator` isso não acontece até que todos os dados sejam devolvidos.

Ainda que o número de retornos produzido com um `function generator` seja igual ao produzido com o `return`, seu uso é mais eficiente, pois, seguindo o exemplo apresentado, ele não mantém os 10 milhões de dados na memória de uma única vez. Pode parecer confuso, então vamos introduzir um pouco de código para explicar isso.

A base de um `function generator` são as instruções `yield` e `next`. A primeira possui um papel similar ao `return` e faz a devolução de um dado de dentro da função para quem a chamou. Falaremos sobre o `next` em breve, vamos antes compreender o uso do `yield` a partir do código a seguir que é uma modificação do anterior:

```
def gerar_numeros():
    for numero in range(10000000):
        yield numero

numeros_gerados = gerar_numeros()
print( type(numeros_gerados ) )
```

Com o uso da instrução `yield`, a variável `numeros_gerados` não possui mais os 10 milhões de dados. Na realidade, essa variável passa a ser

do tipo `generator object`. Assim, o que temos em nossa posse é um `generator` que será capaz de obter os dados processados pela função, mas sob demanda. Ou seja, ao invés de carregar todos os números de uma única vez, vamos fazer isso aos poucos, à medida que eles forem sendo requisitados no programa.

Imagine a aplicação deste exemplo na sua rede social predileta. É comum que, ao iniciá-la, alguns dados sejam carregados e apresentados no seu feed. Após visualizar todos, há algum botão ou atualização automática, que fará o carregamento de mais dados. Esse é um procedimento similar ao uso de um `function generator`, pois, em vez de carregar todo o seu feed de uma única vez (que poderia ocasionar o processamento de milhares de dados), faz-se esse processo aos poucos.

Voltando a falar sobre o nosso `generator` que está guardado na variável `numeros_gerados`. Precisamos solicitar a ele a geração do dado que desejamos. Isso é feito por meio da função `next` ou utilizando um `for`. Explicaremos o primeiro caso com o exemplo a seguir:

```
def gerar_numeros():
    for numero in range(10000000):
        yield numero
numeros_gerados = gerar_numeros()
um_numero = next(numeros_gerados)
outro_numero = next(numeros_gerados)
print(um_numero)
```

Nesse código, a variável `numeros_gerados` é o nosso `generator`. Ao utilizar a função `next` (linha 5) o `generator` vai devolver o número zero, que é o primeiro dado produzido pela função `gerar_numeros`. A execução subsequente da função `next` (linha 6) fará com que a função produza o número um e o salve na variável `outro_numero`. Chamadas sucessivas à função `next` farão com que a função `gerar_numeros` produza os demais números, um a um, até o 9999999.

Outra característica de um `generator` é que ele é iterável. Ou seja, pode-se utilizar o `for` para percorrer os seus elementos:

```
def gerar_numeros():
    for numero in range(10000000):
        yield numero

for numero_gerado in gerar_numeros():
    print(numero_gerado)
```

No código apresentado, cada iteração do `for` vai obter um dos números gerados pela função `gerar_numeros`, até o seu término. Em geral, o uso dessa estratégia para obtenção dos dados de um `generator` é mais comum que o uso da função `next`.

Ainda que a iteração ocasione a impressão de 10 milhões de números, eles não são armazenados todos de uma única vez na memória. Sabemos que o `for` possui um escopo e que seus dados são eliminados ao término de cada iteração. Portanto, cada número guardado na variável `numero_gerado` deixa de existir na iteração seguinte.

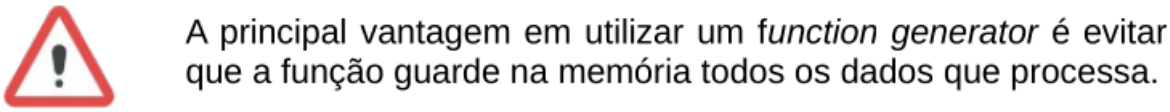

Figura 12.7: A principal vantagem em utilizar um function generator é evitar que a função guarde na memória todos os dados que processa.

O entendimento sobre `generators` passa pela compreensão sobre como o nosso código é executado nessa situação. A seguir apresentamos uma simples função que vai gerar três números e que possui chamadas ao `print` para ilustrar sua sequência de execução:

```
print("Começo do código")

def exemplificar():
```

```
    print("Início da função")
    for numero in range(3):
        print(f"Número sendo produzido {numero}")
        yield numero
    print("Fim da função")

print("Criação do gerador")
gerador_numeros = exemplificar()
print("Chamada do next")
numero = next(gerador_numeros)
numero = next(gerador_numeros)
numero = next(gerador_numeros)

print("Fim do código")
```

A saída apresentada pelo código será:

```
Começo do código
Criação do gerador
Chamada do next
Início da função
Número sendo produzido 0
Número sendo produzido 1
Número sendo produzido 2
Fim do código
```

Observe que o texto *Início da função*, que demonstra quando ela foi executada, somente aparece após a primeira chamada do `next`. Esse é um comportamento diferente do esperado, pois a instrução `gerador_numeros = exemplificar()` não executou o escopo da função que era esperado. Isso acontece com funções que possuem um `yield`, pois nesses casos a sua execução somente ocorrerá quando o `next` for chamado ou o `generator` for iterado.

Outro detalhe importante é que a primeira chamada de `next` inicia o escopo da função, mas as subsequentes não retomam a execução desde o começo. Observe que o primeiro `next` fez aparecer o texto *Início da função* e, em seguida, *Número sendo produzido 0*. A segunda chamada de `next` fez exibir o texto *Número sendo*

produzido 1, pois nesse caso a execução da função é retornada a partir do escopo da repetição (onde parou o último `yield`). Para entender a relação entre `yield` e `next` veja a imagem a seguir que apresenta a execução de duas funções, uma tradicional e outra com `function generator`.

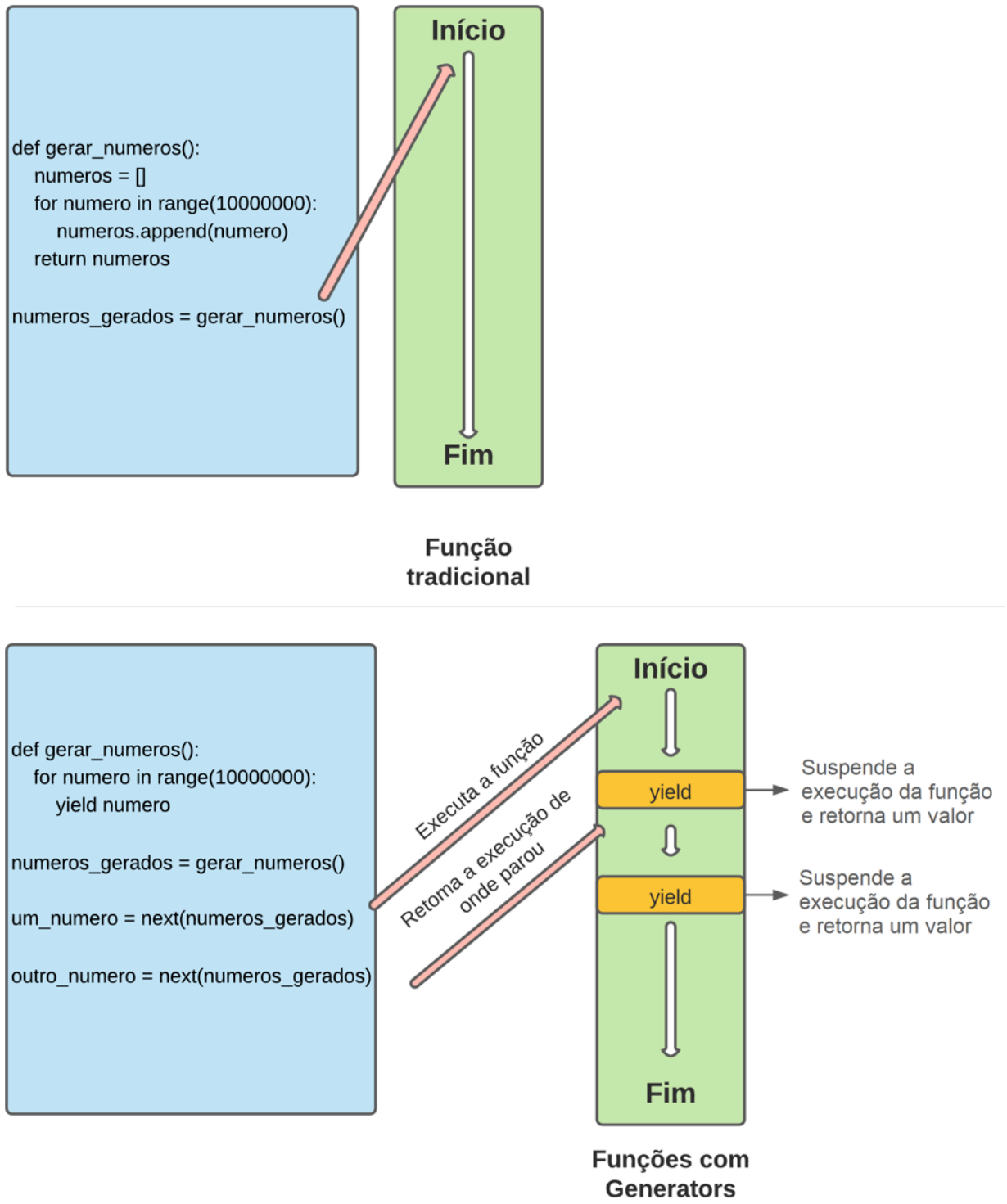

Figura 12.8: Comparação entre uma função normal e outra que utiliza o yield.

A execução de uma função que não utiliza o `yield` ocorre de forma linear, iniciando na primeira linha do seu escopo até a última. Por

outro lado, quando a função possui `yield` (código à direita) seu fluxo de execução é diferente. No primeiro uso do `next`, a função vai executar linearmente até o primeiro `yield` que encontrar. Nesse momento, há a suspensão da execução da função e um valor é retornado. No segundo uso do `next`, a execução retorna do ponto onde parou. No código apresentado, ele vai retornar ao escopo do `for`, pois foi o local do `yield` mais recente.

Podemos realizar diferentes processamentos com os resultados de um `function generator`. Um deles é a possibilidade de criar uma lista a partir de um `generator`, como exemplificado a seguir:

```
def gerar_numeros():
    for numero in range(10000000):
        yield numero

numeros_gerados = gerar_numeros()
lista_numeros = list(numeros_gerados)
```

Também é possível utilizar o conceito de compreensão de listas para criar uma lista a partir de alguma operação em um conjunto de valores de um `generator`. A seguir, vamos criar uma lista a partir de uma multiplicação dos números gerados por `gerar_numero`:

```
def gerar_numeros():
    for numero in range(10000000):
        yield numero

numeros_gerados = gerar_numeros()
lista_numeros = [ numero * 2 for numero in numeros_gerados ]
```

Para iniciantes na programação, o uso de `generators` pode não fazer muito sentido. Não se preocupe, neste momento não é preciso entender por completo todos os tópicos apresentados nesta seção, mas vale saber que esse recurso é importante, em especial para manipulação de grandes volumes de dados que são cada vez mais comuns em nosso dia a dia.

12.3 Tópicos avançados em contêineres

Nos capítulos 10 e 11 apresentamos quatro tipos de contêineres: listas, tuplas, conjuntos e dicionários. Também foi demonstrado um outro uso das listas por meio das matrizes. Esses formatos resolvem boa parte dos seus problemas, mas há ainda outras formas de estruturar os seus dados que serão apresentadas nesta seção, como pilhas e filas. Por fim, há duas funções que são úteis no uso de contêineres: `zip` e `enumerate`, que também serão descritas.

Pilhas e filas

Os contêineres nos possibilitam representar diferentes **estruturas de dados**, que seriam as formas como os dados são melhor organizados para representar uma determinada situação. Há uma variedade de estruturas e nesta seção vamos abordar as pilhas e filas, pois elas estão presentes com frequência em nosso dia a dia e consequentemente são utilizadas nos softwares.

Para exemplificar uma fila, lembre-se da experiência de ir ao banco pagar uma conta. Há uma chance de que outras pessoas tenham chegado antes de você e que uma **fila** tenha sido formada para controlar o atendimento. Nesse formato, o primeiro cliente que chegou será o primeiro a ser atendido e assim sucessivamente até a pessoa que chegou por último. Em inglês nomeamos essa característica das filas como *First in, first out (FIFO)*, ou seja, o primeiro que chegou é o primeiro a sair.

A estrutura de dados do tipo pilha é utilizada para representar uma situação de "empilhamento". Por exemplo, considere que após um almoço cada pessoa coloca o seu prato sujo na pia da cozinha, um em cima do outro, realizando um empilhamento. Na hora de lavá-los, o primeiro a ser retirado será o que está no topo da pilha, ou seja, o que foi colocado por último. Esse procedimento continua até não existirem mais pratos. Neste formato dizemos que o último que chegou será o primeiro a sair, em inglês *Last in, First out (LIFO)*.

Em uma estrutura de dados do tipo fila, o primeiro elemento adicionado será o primeiro a ser removido.

Figura 12.9: Em uma fila, o primeiro elemento adicionado será o primeiro a sair.

Na prática, as filas e pilhas podem ser representadas por meio de listas. Isso é possível, pois elas posicionam seus elementos por meio de índices ordenados, o que torna possível saber quem é o primeiro (índice 0) e o último elemento (índice n-1).

Em uma estrutura de dados do tipo pilha, o primeiro elemento adicionado será o último a ser removido.

Figura 12.10: Em uma pilha, o último elemento adicionado será o primeiro a sair.

Para representar o comportamento FIFO ou LIFO devemos seguir algumas regras para inserir e acessar os dados guardados. É importante que os elementos sejam inseridos em uma ordem correta, representando a disposição da fila ou pilha. O que vai mudar é a forma como vamos acessar os dados guardados.

Em geral, após acessar um elemento nós o removemos da lista e isso pode ser feito com a função `pop`. Essa função apresenta duas sintaxes:

```
elemento = pop()
```

Nesse caso, o último elemento da lista será removido e retornado à variável `elemento`.

```
elemento = pop( índice )
```

A segunda sintaxe possibilita remover o elemento de um índice específico e o retorna para a variável `elemento`.

Vamos utilizar as duas sintaxes para acessar os elementos em uma fila e pilha.

Fila

Nas filas, o elemento de índice zero será o primeiro a ser recuperado. Após ele sair da lista, o segundo elemento passará a ser o primeiro. Essa informação é importante, pois determina a forma como utilizaremos a função `pop` para acessar os elementos de uma fila.

No exemplo a seguir temos uma variável para representar a fila de um banco e vamos recuperar os seus elementos seguindo a ordem do primeiro para o último.

```
fila_banco = ["Leonardo", "Gabriel", "Davi", "Pietra" ]

primeiro_atendimento = fila_banco.pop(0) # Recupera Leonardo
segundo_atendimento = fila_banco.pop(0) # Recupera Gabriel
terceiro_atendimento = fila_banco.pop(0) # Recupera Davi
quarto_atendimento = fila_banco.pop(0) # Recupera Pietra

print(fila_banco)
```

Para cada vez que a função `pop(0)` for chamada, um elemento é removido da lista e retornado para uma variável. Como o parâmetro informado foi zero, ele vai sempre remover o primeiro elemento. O elemento que era o segundo da lista passa a ser o primeiro e será acessado na chamada subsequente a `pop(0)` e assim sucessivamente. Ao término da execução do código apresentado, o

`print` apresentará uma lista vazia, pois todos os elementos foram recuperados.

Atenção. O código apresentado é funcional e resolve a maior parte dos problemas que vão surgir. No entanto, como a função `pop` não é eficiente para remoção constante de elementos que estão no início ou meio da lista, o Python recomenda o uso do tipo `deque` para implementação de filas que possuam um grande número de elementos. Maiores detalhes podem ser encontrados na documentação oficial no link https://docs.python.org/pt-br/3/tutorial/datastructures.html#using-lists-as-queues.

A inserção de elementos em uma fila ou pilha já criada deve ser feita por meio da função `append`, pois ela sempre aloca o dado na última posição. A função apresenta a sintaxe:

```
nome-da-lista.append( dado )
```

O uso da função é apresentado a seguir:

```
fila_banco = ["Leonardo", "Gabriel", "Davi", "Pietra" ]
fila_banco.append("Linus")

print(fila_banco)
```

O código resultará na saída: `['Leonardo', 'Gabriel', 'Davi', 'Pietra', 'Linus']`.

Pilha

Para as pilhas, o primeiro elemento a ser recuperado é o que se encontra no último índice da lista (n-1). Após a sua remoção da lista, o elemento anterior passará a ser o último.

Vamos exemplificar esse procedimento por meio de um código que implementa a operação de *desfazer (ctrl + z)*, que frequentemente é feita por meio de uma pilha. Vamos apresentar um cenário de uso

em um editor de textos, onde quatro operações foram realizadas por um editor, da mais antiga para a mais recente: texto em negrito, texto em itálico, mudança de fonte, mudança no tamanho da fonte. Uma pilha é utilizada para agrupar essas informações e para saber a ordem em foram realizadas (lembre-se de que as ações são desfeitas de trás para frente, ou seja, da mais recente para a mais antiga).

```
comandos = ["negrito", "itálico", "mudança_fonte"]

comandos.append("mudança_tamanho_fonte")

acao_recente = comandos.pop() # vai recuperar mudança_tamanho_fonte
segunda_acao_recente = comandos.pop() # vai recuperar mudança_fonte
terceira_acao_recente = comandos.pop() # vai recuperar itálico
quarta_acao_recente = comandos.pop() # vai recuperar negrito
```

Como as pilhas e filas são contêineres do tipo listas, compete à pessoa programadora a responsabilidade de utilizar a função `pop` e `append` da forma adequada para representar os comportamentos destes recursos.

Funções zip e enumerate

A função `zip` é utilizada para agregar os dados de múltiplos iteráveis (como listas, tuplas, entre outros), criando um iterador único. Vamos explicar, considere a figura a seguir, onde temos duas listas (parte superior) e o resultado da operação `zip` (parte inferior).

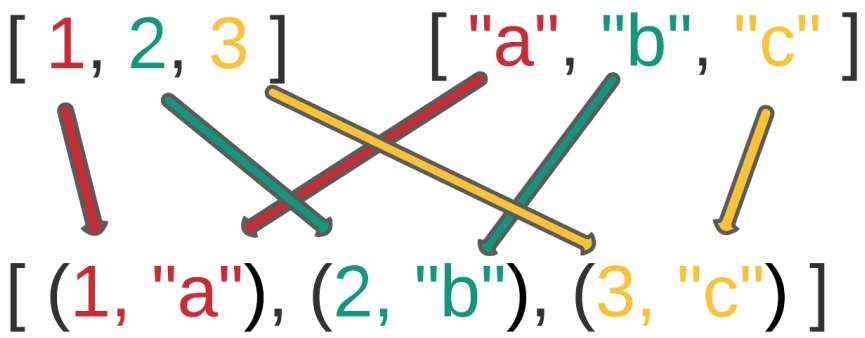

Figura 12.11: Iterador criado com o zip.

Observe como o `zip` cria uma lista formada por pares de tuplas. Esses são construídos pela junção dos elementos de uma tupla com a de outra. Assim, o primeiro elemento da lista um será unido ao primeiro elemento da lista dois. O segundo elemento da lista dois será unido ao segundo da lista dois, e assim sucessivamente.

A sintaxe para uso da função `zip` está representada a seguir:

```
iterador-criado = zip( iteraveis )
```

A função `zip` recebe zero ou mais iteráveis como parâmetros. A depender do que é informado, então o retorno será diferente. Caso não seja informado nenhum iterador, então o retorno é um iterador vazio. Ao utilizar apenas um parâmetro, retorna-se um iterador formado por tuplas de um único elemento, como no código a seguir:

```
numeros = [ 1, 2, 3 ]
retorno_zip = zip( numeros )
zip_convertido_em_lista = list(retorno_zip)
print(zip_convertido_em_lista)
```

Antes de falar sobre o resultado do código apresentado é preciso entender o que ocorreu na linha 3. Com frequência você vai transformar o resultado de `zip` em um contêiner. Nesse exemplo o convertemos para uma lista com a função `list`, fazendo com que o nosso código apresente a saída: `[(1,), (2,), (3,)]`

Com a passagem de dois parâmetros teríamos o seguinte código:

```
numeros = [ 1, 2, 3 ]
letras = [ "a", "b", "c" ]
retorno_zip = zip( numeros, letras )
zip_convertido_em_lista = list(retorno_zip)
print(zip_convertido_em_lista)
```

A execução do código resultará na saída: `[(1, 'a'), (2, 'b'), (3, 'c')]`, como apresentado na figura anterior. Essa lógica se aplica para a passagem de mais de dois parâmetros:

```
numeros = [ 1, 2, 3 ]
letras = [ "a", "b", "c" ]
flutuantes = [ 0.5, 0.8, 10.3 ]
retorno_zip = zip( numeros, letras )
zip_convertido_em_lista = list(retorno_zip)
print(zip_convertido_em_lista)
```

A saída será: `[(1, 'a', 0.5), (2, 'b', 0.8), (3, 'c', 10.3)]`

Nos exemplos apresentados os iteradores passados como parâmetro de `zip` possuíam a mesma quantidade de elementos. Em casos onde há iteradores de diferentes tamanhos, o resultado de `zip` terá como base o menor tamanho, fará às junções e descartará os demais elementos os iteradores com mais elementos. Veja um exemplo a seguir:

```
numeros = [ 1, 2, 3, 4, 5 ]
letras = [ "a", "b", "c" ]
retorno_zip = zip( numeros, letras )
zip_convertido_em_lista = list(retorno_zip)
print(zip_convertido_em_lista)
```

O resultado desse código será: `[(1, 'a'), (2, 'b'), (3, 'c')]`. Como o iterador `letras` possuía tamanho 3 e o iterador `numeros`, tamanho 5, então o `zip` utilizou apenas os três primeiros elementos deste último e descartou os demais.

Você pode estar se perguntando da utilidade do `zip`. Vamos demonstrar duas, a primeira é criar dicionários a partir de duas listas ou tuplas diferentes. Lembre-se de que um dicionário é formado por associação entre chaves e valores. Assim, utilizando duas listas, os elementos da primeira servirão de chave e os elementos da segunda serão seus valores. O código a seguir apresenta uma lista de CPFs e outra lista de nomes. Elas serão combinadas para formar um dicionário:

```
cpf = [ 54254, 10944, 45679 ]
nomes= [ "LinusTorvalds", "SteveWozniak", "GuidoRossum" ]
retorno_zip = zip( cpf , nomes)
dicionario = dict(retorno_zip )
print(dicionario)
```

Observe como utilizamos a função `dict` para criar um dicionário a partir do iterador construído em `zip`. Assim, o resultado deste código será o dicionário: `{54254: 'LinusTorvalds', 10944: 'SteveWozniak', 45679: 'GuidoRossum'}`.

Um outro uso da função `zip` é permitir a iteração por duas listas ou tuplas simultaneamente. Esse processo é realizado por meio de um `for` e apresenta a sintaxe:

```
for elemento_iterador_1, elemento_iterador_2 in zip( iterador_1, iterador_2)
```

Lembre-se de que a função `zip` produz tuplas formadas a partir da junção entre os elementos dos diferentes iteradores. Assim, em cada repetição do `for` teremos acesso aos dados dos diferentes iteradores, um por vez, que estarão disponíveis nas variáveis

`elemento_iterador_1` e `elemento_iterador_2`. O código a seguir demonstra como realizar esse processo:

```
numeros = [ 1, 2, 3, 4, 5 ]
letras = [ "a", "b", "c" ]
for numero, letra in zip(numeros, letras):
    print(f"Elemento primeiro iterador: {numero}")
    print(f"Elemento segundo iterador: {letra }")
```

E resultará na saída:

```
Elemento primeiro iterador: 1
Elemento segundo iterador: a
Elemento primeiro iterador: 2
Elemento segundo iterador: b
Elemento primeiro iterador: 3
Elemento segundo iterador: c
```

Caso sejam utilizados mais de dois iteradores no `zip`, basta acrescentar a variável na instrução do `for` e separá-la com vírgula.

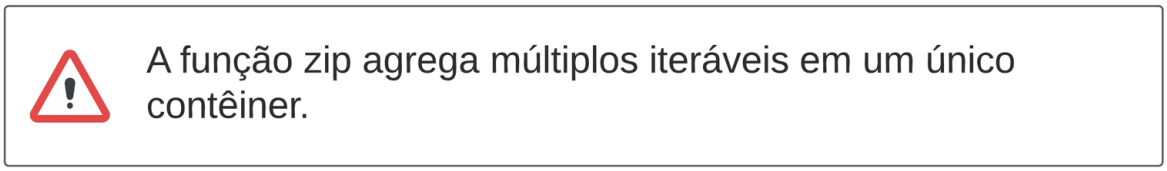

Figura 12.12: A função zip possibilita a agregação de múltiplos iteráveis em um único contêiner.

Antes de explicar o uso da função `enumerate` vamos compreender a sua importância por meio do código a seguir. Nele, temos uma lista com alguns números. Vamos iterar sobre ela e a cada dois números exibir uma mensagem. Ou seja, considerando que a iteração começa em 0, número par, vamos apenas mostrar a mensagem para as iterações de ímpares.

```
contador = 0
```

```
numeros = [ 1, 0, 4, 5, 9, 10, 3, 7 ]

for numero in numeros:
    if contador % 2 != 0:
        print("Exibir uma mensagem")
    contador += 1
```

Observe que temos uma variável nomeada `contador` que é utilizada para contabilizar em qual iteração nosso código se encontra. Essa informação é necessária, pois é utilizada para sabermos quando a iteração era ímpar (critério para exibir uma mensagem). Ainda que essa estratégia funcione, caso o(a) programador(a) esqueça de incrementar a variável contadora (última linha do código) teríamos um grande problema. Para evitar problemas como esse, podemos fazer uso do `enumerate`. Vamos explicar.

A função `enumerate` realiza uma contagem incremental para cada iteração em um `for`. Seu uso é realizado por meio da sintaxe:

for variavel-contadora, dado in enumerate(iterador):

O parâmetro passado para a função é um iterável, que pode ser uma lista, *range*, dicionário, entre outros. Quando essa função é utilizada em um `for`, para cada iteração será retornado um elemento do iterável juntamente com o contador que inicia em zero e é incrementado em cada repetição.

Vamos visualizar seu uso com uma modificação do código anterior e em seguida explicar sua sintaxe.

```
numeros = [ 1, 0, 4, 5, 9, 10, 3, 7 ]

for contador, numero in enumerate(numeros):
    if contador % 2 != 0:
        print("Exibir uma mensagem")
```

O resultado produzido por este código é similar ao anterior. No entanto, observe que não precisamos nos preocupar com a variável contadora, pois o próprio `enumerate` cuidará disto para nós.

Empacotamento e desempacotamento

Os operadores `*` e `**`, conhecidos no Python por **unpacking operators**, foram apresentados quando falamos sobre quantidade variável de parâmetros em funções. Eles também podem ser utilizados em outros cenários, como apresentado a seguir.

No primeiro exemplo, temos uma função que recebe múltiplos parâmetros, mas no momento da sua execução apenas um será passado. Isso é possível por meio do operador `*` e do uso de uma lista, tupla ou conjunto. Vamos visualizar o exemplo:

```
def somar(numero_um, numero_dois, numero_tres):
    return numero_um + numero_dois + numero_tres
numeros_soma = [ 5, 8, 10 ]
resultado = somar(*numeros_soma)
```

Observe que na declaração da função `somar` ela recebe três parâmetros (linha 1), mas em sua chamada apenas passamos uma variável (linha 4). Isso é possível com o uso do operador `*` e uma variável do tipo tupla, lista ou conjunto que contenha o mesmo quantitativo de elementos que o número de parâmetros esperado pela função. Assim, o Python vai extrair cada valor e atribuir a um dos parâmetros.

Para listas e tuplas, o Python seguirá a ordem dos elementos, pois esses contêineres possuem um índice ordenado. No exemplo, a variável `numero_um` guardará o valor 5, `numero_dois`, o valor 8, e assim sucessivamente. No entanto, se um conjunto for utilizado não há essa garantia, pois seus elementos não possuem uma ordem. Assim, não há como saber para qual variável um determinado valor será atribuído. Se isso for importante, então não se deve utilizar este tipo de dado.

O operador `**` também pode ser aplicado na passagem de um dicionário como parâmetro de função. Nesse caso, o Python utilizará as chaves do dicionário para definir qual parâmetro receberá os valores. Explicaremos esse ponto modificando o exemplo anterior:

```
def somar(numero_um, numero_dois, numero_tres):
    return numero_um + numero_dois + numero_tres
numeros_soma = { "numero_tres" : 10, "numero_um" : 5, "numero_dois" : 8 }
resultado = somar(**numeros_soma)
```

Observe que neste exemplo a variável `numeros_soma` é um dicionário. Esse dado é passado como parâmetro da função `somar` por meio do operador `**`. Uma restrição para a realização desta operação é que as chaves do dicionário devem ser iguais aos nomes dos parâmetros da função, caso contrário, será lançado o erro `NameError`.

As listas e tuplas também permitem extrair seus elementos e atribuí-los a múltiplas variáveis com uma única instrução. Esse procedimento é realizado com a seguinte sintaxe:

```
variavel_um, variavel_dois, variavel_n = variavel-tipo-tupla
```

Onde, as variáveis do lado esquerdo precisam ser de igual quantitativo ao número de elementos da lista. O Python irá atribuir à cada uma delas, um elemento da lista, seguindo a ordem dos seus índices. Observe o código a seguir:

```
partes_computador = ["memória", "processador", "disco"]
parte_um, parte_dois, parte_tres = partes_computador
```

Com a execução deste código, a variável `parte_um` terá como valor a *String* memória, `parte_dois`, a *String* processador, e assim sucessivamente.

12.4 Tratamento de erros no Python

No capítulo 5 aprendemos sobre o que são erros na programação e situações que ocasionam esse problema. É importante relembrar que quando erro é encontrado no código, as instruções subsequentes àquela que o ocasionou não serão executadas e o nosso programa encerra abruptamente. Claramente não é uma situação desejada.

O código a seguir ilustra um dos erros do Python, ocasionado pela falta de aspas na declaração de uma variável do tipo *String*:

```
pais = "Brasil
print(pais)
```

A execução do código fará com que o interpretador do Python lance (sim, este é o termo utilizado) um erro do tipo `SyntaxError` e consequentemente a instrução da segunda linha não será executada. O problema apresentado decorre da violação de uma das regras da linguagem de programação, fato conhecido por **erro de sintaxe**.

Erros de sintaxe são mais simples e editores de programação mais avançados, como o *Visual Studio Code* (https://code.visualstudio.com/) e o *PyCharm* (https://www.jetbrains.com/pt-br/pycharm/), conseguem detectá-los enquanto você escreve o seu código. Portanto, sugerimos fortemente o uso desses editores para facilitar o seu trabalho enquanto programador(a).

Há outros tipos de erros que são ocasionados pelo uso incorreto de recursos da linguagem de programação, mas que ocorrem apenas durante a execução do programa. Um exemplo é o *KeyError*, lançado quando se tenta utilizar chaves inexistentes de um dicionário. Nestes casos, o código não apresenta nenhum erro de sintaxe e os editores acima mencionados não vão apontar para nenhum problema. Entretanto, durante a execução do código o erro pode ser ocasionado.

No código a seguir ilustramos um caso de *KeyError*, com a criação de um dicionário que representa informações de um veículo. Em seguida, pedimos ao usuário da aplicação que informe qual dado do veículo deseja visualizar. A informação digitada pelo usuário será utilizada para localizar o dado no dicionário. Como o erro mencionado apenas ocorre na inexistência da chave no dicionário, vamos considerar um cenário onde o usuário solicita visualizar o ano do veículo, um dado que não está descrito no dicionário:

```
veiculo = { "marca" : "Volkswagen", "modelo": "Fusca" }
dado = input("Qual dado deseja visualizar do veículo?")
informacao_veiculo = veiculo[dado]
print(informacao_veiculo)
print("Obrigado por utilizar o programa")
```

Observe que o código não apresenta problemas de sintaxe. Quando o usuário digitar *ano*, um erro do tipo `KeyError: 'ano'` será lançado. Encerrar o programa não é desejado e o ideal é oferecer uma resposta ao usuário sobre o problema. Para isso, precisamos realizar um procedimento que envolve um **tratamento do erro**.

O tratamento de erros é uma forma de evitar o encerramento do programa na ocorrência do erro, possibilitando a implementação de instruções para reverter a situação e/ou apresentar alguma mensagem ao usuário sobre o ocorrido.

Para utilizar o tratamento de erros é preciso identificar os trechos de código que **podem** ocasionar erros de programação. Isso requer um conhecimento sobre quais instruções ocasionam erros, algumas já mencionadas neste livro e outras descritas na documentação oficial do Python (https://docs.python.org/). O segundo passo é utilizar instruções de programação para capturar o erro (utilizamos esse termo para indicar que estamos interessados em ser avisados da sua ocorrência) e, por fim, definir a resposta que deve ser dada nessa situação. Vamos analisar esses pontos separadamente.

Identificar os trechos de código que podem ocasionar erros é uma tarefa difícil e que requer profundo conhecimento da linguagem da

programação. É preciso saber quais funções e/ou instruções podem lançar erros e em quais situações. No exemplo anterior, a instrução presente na linha 3 **pode** ocasionar um erro quando a chave informada não existir no dicionário. Destacamos a palavra *pode*, pois se o usuário digitar *marca* ou *modelo*, chaves presentes no dicionário, então a execução do programa ocorrerá sem nenhum problema. Por essa razão que há erros que somente são identificados em **tempo de execução**, pois ocorrem quando o programa está sendo utilizado.

Conhecendo os trechos do código que podem ocasionar os erros em tempo de execução, devemos utilizar um recurso do Python para capturá-los e dar a resposta adequada. Quando falamos em resposta, significa pensar em como o seu programa deve reagir à ocorrência do problema. Por exemplo, no código anterior seria mais adequado exibir uma mensagem ao usuário e informá-lo que a informação que tentou acessar não existe.

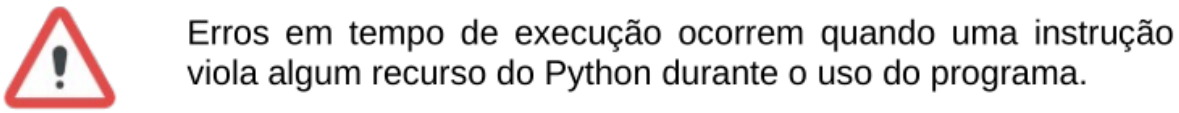

Figura 12.13: Erros em tempo de execução ocorrem quando uma instrução viola algum recurso do Python durante o uso do programa.

O processo de detecção de erros é feito com uso da instrução `try`. Sua sintaxe é representada por:

```
try:
    # escopo com as instruções que lançam erros.
```

Observe que o `try` possui um escopo. Dentro dele devemos informar apenas os trechos de código que podem lançar erros e as instruções subsequentes que utilizam a informação da instrução lançadora de erros. Explicaremos esses pontos com o código a seguir, uma modificação do anterior para representar o uso de `try`:

```
veiculo = { "marca" : "Volkswagen", "modelo": "Fusca" }
dado = input("Qual dado deseja visualizar do veículo?")
try:
    informacao_veiculo = veiculo[dado]
    print(informacao_veiculo)
```

Observe as duas instruções no escopo do `try`. A primeira pode ocasionar um *KeyError*, enquanto que a segunda somente fará sentido de existir caso a execução da anterior seja concluída com sucesso. Por essas razões, ambas estão no escopo.

Não há um limite na quantidade de tratamentos de erro que o seu código pode ter ou para a quantidade de instruções no escopo do `try`. No entanto, como descrito anteriormente, é comum incluir apenas as instruções associadas à ocorrência do erro.

A definição da resposta de um erro é conhecida por **tratamento de erro** e utiliza a instrução `except`. Ela também possui um escopo, onde escreveremos os códigos responsáveis por realizar algum procedimento quando um erro acontece. Vamos ver o exemplo completo:

```
veiculo = { "marca" : "Volkswagen", "modelo": "Fusca" }
dado = input("Qual dado deseja visualizar do veículo?")
try:
    informacao_veiculo = veiculo[dado]
    print(informacao_veiculo)
except:
    print("O dado que tentou visualizar não existe no veículo")

print("Obrigado por utilizar o programa")
```

Caso um erro seja ocasionado nos códigos presentes no `try`, então a execução do programa é transferida para o escopo do `except`. Após a execução das instruções do `except` o programa dará seguimento as demais. Por essa razão, independente do que aconteça (com ou sem erros), o último `print` será executado e apresentará a mensagem *Obrigado por utilizar o programa*.

A instrução *try* é utilizada para definir o escopo de código que pode ocasionar um erro. A resposta para ele é feita por meio da instrução *except*.

Figura 12.14: A instrução try é utilizada para executar instruções que podem ocasionar erros. As instruções de resposta para os erros são definidas no escopo de except.

Se houver outras instruções dentro do escopo de `try` que ocasionam erros, eles também serão capturados. No entanto, há situações onde desejamos capturar erros específicos, por exemplo, apenas implementar uma resposta para erros do tipo *KeyError* e ignorar os demais. Isso é possível, como também lançar nossos próprios erros. No entanto, esses são temas mais complexos e sugerimos a leitura da documentação oficial do Python (https://docs.python.org/pt-br/3/tutorial/errors.html).

12.5 Quais os próximos passos?

Chegamos ao término da nossa jornada sobre a introdução à programação com Python. Se você completou o livro na sequência que sugerimos, aprendeu os conceitos básicos de qualquer programa ou jogo eletrônico e ficamos felizes com isso! :)

Até se tornar programador profissional há um longo caminho a seguir, outros conhecimentos que deve possuir, como também definir qual área pretende se especializar. Isso é necessário, pois além dos tópicos básicos de programação há outros que compõem um programa, como representado, de forma simplificada, na figura a seguir.

Figura 12.15: Tópicos que compõem um programa.

Devemos interpretar a pirâmide apresentada como um empilhamento de conhecimentos que muitas vezes são dependentes. Por exemplo, a Orientação a Objetos utiliza muito dos conteúdos abordados em introdução à programação, assim como os conhecimentos especializados também fazem uso de ambos. Por isso acreditamos que ao seguir uma sequência de estudos o seu aprendizado é facilitado.

Em nossa opinião, Orientação a Objetos deve ser o próximo assunto a ser estudado. Ele é o que chamamos na computação de *paradigma* e o que define a forma como um problema é representado na programação. Existem diversos paradigmas, mas há 20 anos este ainda é o mais utilizado e por isso recomendamos o seu aprendizado. Se este é o seu primeiro contato com a programação, sugerimos que permaneça no Python para aprender sobre este paradigma. Após isso, você pode (e deve) explorar outras linguagens de programação.

A seguir, temos conhecimentos especializados que também são aplicados na criação de um programa. Nesta lista entram diversos tópicos, como estrutura de dados (avançam na compreensão sobre outras formas de representar dados para além das filas e pilhas, como o uso de árvores binárias, tabelas hash, entre outros), padrões e antipadrões de projeto, testes de software, banco de dados.

A pessoa programadora também precisa definir a tecnologia onde o software será desenvolvido. Por exemplo, jogos eletrônicos requerem o conhecimento de ferramentas, como o *PyGame*, *Unit*, entre outros. A criação de aplicações para a Web vai demandar o aprendizado de tecnologias nessa área, como o Django, Flask, ou mesmo de outras linguagens de programação que são amplamente adotadas para este fim, como o JavaScript. Em muitos casos a escolha dessas tecnologias não é de quem programa, mas sim da empresa na qual ela trabalha. Ainda assim, como aprendiz você pode estudá-las para ampliar seus conhecimentos sobre o tema.

Para finalizar, sabemos que aprender a programar não é uma tarefa fácil que exige resiliência e autoconfiança para superar as dificuldades que ocorrem ao longo da jornada. Além disso, é preciso encarar o aprendizado como um processo contínuo, não apenas pela existência de uma variedade enorme de tecnologias e linguagens de programação a serem exploradas, como pelo fato de elas evoluírem, fazendo surgir novidades com frequência.

Acreditamos que os resultados do aprendizado recompensam, não apenas financeiramente, mas também por instigar em nós a capacidade de resolver problemas, o que é bem legal! :)

Desejamos sucesso em sua jornada na programação e ficamos felizes por fazer parte dela!

Atenciosamente,

Leonardo Soares e Silva

Gabriel Fortes de Macêdo

Capítulo 13
Referências

1. BRASSCOM - Associação Brasileira das Empresas de Tecnologia da Informação e Comunicação. Formação Educacional e Empregabilidade em TIC Achados e Recomendações: relatório técnico. São Paulo, 2019.

2. DE JESUS, Emanuel. Teaching computer programming with structured programming language and flowcharts. In: *Proceedings of the 2011 Workshop on Open Source and Design of Communication*. 2011. p. 45-48.

1. DOUKAKIS, Dimitrios; GRIGORIADOU, Maria; TSAGANOU, Grammatiki. Understanding the programming variable concept with animated interactive analogies. In: *Proceedings of the The 8th Hellenic European Research on Computer Mathematics & Its Applications Conference* (HERCMA'07). 2007.

2. GOMES, Anabela; MENDES, António José. Problem Solving in Programming. In: *PPIG*. 2007. p. 18.

3. GUARDA, Graziela Ferreira; PINTO, Sérgio Crespo CS. Dimensões do Pensamento Computacional: conceitos, práticas e novas perspectivas. In: *Anais do XXXI Simpósio Brasileiro de Informática na Educação*. SBC, 2020. p. 1463-1472.

4. NORMAN, Donald A. (Ed.). *Models of human memory*. Elsevier, 2013.

5. PRATHER, James; PETTIT, Raymond; BECKER, Brett; DENNY, Paul; DASTYNI, Loksa; PETERS, Alani; ALBRECHT, Zachary; MASCHI, Krista. First things first: Providing metacognitive scaffolding for interpreting problem prompts. In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. 2019. p. 531-537.

6. PUTNAM, Ralph T. et al. A summary of misconceptions of high school Basic programmers. *Journal of Educational Computing Research*, v. 2, n. 4, p. 459-472, 1986.

7. REALPYTHON. 8 World-Class Software Companies That Use Python. Disponível em: https://realpython.com/world-class-companies-using-python/. Acesso em: 18/10/2020.

8. SAFEI, Suhailan; SHIBGHATULLAH, Abdul Samad; MOHD ABOOBAIDER, Burhanuddin. A perspective of automated programming error feedback approaches in problem solving exercises. *Journal of Theoretical and Applied Information Technology*, v. 70, n. 1, p. 121-129, 2014.

9. WANG, Yingxu; CHIEW, Vincent. On the cognitive process of human problem solving. *Cognitive systems research*, v. 11, n. 1, p. 81-92, 2010.